Penerapan Algoritma BFS untuk Menentukan Langkah Terpendek Mendapatkan *4-orb* dalam Permainan Sdorica

Hanif Muhamad Gana 13518127
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
13518127@std.stei.itb.ac.id

***Abstrak*—Sdorica adalah sebuah *game fantasy strategy rpg* yang berfokus pada petarungan antar karakter dengan *skill* yang unik. *Skill* tersebut diaktifkan dengan menggunakan *orb* pada *command zone*. *Skill* yang diaktifkan dengan *4-orb* umumnya merupakan *skill* paling kuat, tapi mendapatkan *4-orb* bisa memerlukan banyak *turn* dan memberikan kesempatan lawan untuk menyerang. Oleh karena itu diperlukan cara optimisasi setiap turn supaya bisa menemukan langkah terpendek menuju *4-orb*. Karena merupakan masalah optimisasi, masalah ini bisa dibantu dengan menggunakan algoritma BFS.**

Kata Kunci—turn; optimisasi; Sdorica; 4-orb

I. PENDAHULUAN

Permainan pada dasarnya merupakan sesuatu yang dipakai untuk bersenang-senang. Permainan banyak sekali macamnya dan sudah ada sejak dahulu kala. Sepak bola, kelereng, kartu, *table-top*, melempar batu, semua itu dapat disebut sebagai permainan.

Salah satu jenis permainan yang cukup baru dan pesat perkembangannya saat ini adalah *video game* atau 'permainan vidio'. Jenis permainan merupakan permainan yang menggunakan tampilan monitor untuk menunjukkan gambar-gambar interaktif dari permainan tersebut. *Video game* pertama yang dapat dimainkan publik adalah Pong pada tahun 1970-an, meskipun sebelum-sebelumnyanya ada permainan lain yang dapat dikategorikan sebagai *video game* yang tidak dirilis ke publik.

Video Game pada awalnya hanya menampilkan kotak-kotak '*pixel'* yang merepresentasikan keadaan permainannya. Namun, seiring berjalannya waktu *video game* mulai dapat menunjukkan warna dan gambar-gambar lain yang lebih kompleks daripada kotak-kotak *'pixel'* saja dan bahkan sudah bisa melompat dari hanya menunjukkan gambar-gambar dalam tampilan 2D menjadi model-model 3D. Meskipun begitu, tampilan permainan dalam 2D tetap banyak diminati dan banyak *video game* yang menggunakan tampilan 2.5D – yaitu permainannya sebenarnya 3D dan memiliki model-model akan tetapi ditunjukkan selayaknya *video game* 2D.

Selain dibedakan dalam cara menampilkan aspek *video* dari *video game*, *video game* juga dibedakan dari genre *game* yang dimilikinya. Sama seperti media hiburan lain, seperti film dan musik, *video game* memiliki jumlah *genre* yang sangat banyak dan beragam. Misalnya saja genre *role-playing*, strategi, *adventure*, *platformer*, dan lain sebagainya.

Genre permainan *role-playing (RPG)* adalah jenis permainan yang memungkinkan pemainnya unuk melakukan berbagai misi, umumnya disebut *quest*, untuk mendapatkan *experience* bagai karakter pemain dan menjadi lebih kuat. *Genre* permainan ini hampir seluruhnya didefinisikan oleh sebuah permainan yang bernama *Dungeons & Dragons*. Seperti permainan yang mendifinisikannya, *genre* ini pada umumnya bertempat di dunia fantasi dengan para karakter pemainnya melawat suatu kekuatan jahat.

Sementara itu, *genre* permainan strategi adalah jenis permainan yang mementingkan perencanaan secara strategis atau taktis untuk dapat menang, berbeda dengan *video game* lainnya yang umumnya lebih mementingkan refleks dana *skill*. *Genre* ini terbagi menjadi *turn-based strategy* dan *real-time strategy*. Pada permainan *turn-based strategy* setiap pemain memiliki waktu tersendiri untuk merencanakan gerakan selanjutnya, dan setiap pemain akan saling menunggu hingga gilirannya datang untuk beraksi. Sementara itu pada permainan *real-time strategy* setiap pemain bergerak pada waktu yang bersamaan dan pemain harus dapat menyesuaikan rencanya secara cepat.

Sdorica adalah sebuah *video game* fantasy strategy RPG ciptaan Rayark yang dapat dimainkan pada perangkat berbasis Android atau IOS. Permainan Sdorica ini memiliki beragam misi yang dapat meningkatkan karakter-karakter pemain supaya bisa menjadi lebih kuat. Dalam misi-misi ini karakter pemain akan bertarung menggunakan aturan yang mirip dengan *turn-based strategy* dengan menggunakan *command zone* untuk menuntun sebuah karakter pada tim pemain untuk melakukan suatu serangan.

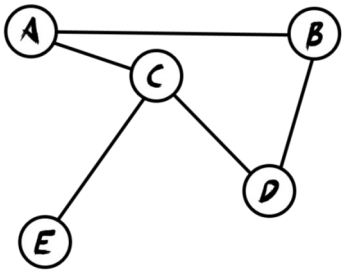
Setiap pemain akan dapat membawa maksimal tiga karakter untuk timnya di suatu misi. Setiap karakter memiliki *skill* yang beragam dan diaktifkan dengan moncocokan *orb* pada *command zone* dengan jumlah tertentu dengan warna

yang sesuai dengan posisi karakter tersebut. Pada umumnya *skill* yang diaktifkan dengan empat *orb* adalah *skill* terkuat karakter. Oleh karena itu penting untuk bisa mendapatkan empat *orb* untuk bisa mengaktifkan *skill* tersebut dengan jumlah *turn* paling rendah supaya musuh tidak mendapatkan kesempatan menyerang yang banyak.

Karena jumlah langkah yang diambil untuk menemukan jawaban penting, dengan semakin sedikit semakin baik, maka persoalan ini dapat dianggap sebagai persoalan optimisasi. Mengingat Breadth First Search lebih cocok untuk persoalan optimisasi dibandingkan Depth First Search maka algoritma yang digunakan adalah algoritma Breadth First Search.

II. Landasan Teori

A. *Graf*



Gambar 2.1 Contoh Sebuah Graf
(Sumber : Dokumentasi penulis)

Sebuah graf dapat digunakan untuk merepresentasikan objek-objek diskrit serta hubungan antara objek-objek tersebut. Sebuah graf pada umumnya terdiri dari sejumlah simpul-simpul yang merepresentasikan objek-objek diskrit tersebut, serta sekumpulan sisi yang menghubungkan simpul-simpul tersebut yang merepresentasikan hubungan antara objek-objek diskrit tersebut.

B. *Breadth First Search*

Breadth First Search (BFS) adalah sebuah algoritma yang melakukan traversal terhadap graf, yaitu mengunjungi simpul-simpul yang ada pada graf secara sistematik. Sebenarnya algoritma BFS ini memiliki saudara yang serupa yaitu Depth First Search (DFS). Perbedaan dari keduanya terletak pada cara mereka melakukan traversal tersebut. Algoritma BFS akan mengunjungi simpul-simpul yang jarak simpulnya dari simpul awal terdekat terlebih dahulu. Sementara itu DFS akan mengunjungi salah satu simpul yang terhubung ke simpul awal dan mengunjungi seluruh simpul yang terhubung dengan simpul tersebut kian mendalam hingga 'habis' sebelum akhirnya kembali ke simpul awal dan mengunjungi simpul lain yang terhubung ke simpul awal dan melakukan hal yang serupa. Oleh karena itu apabila ingin menyelesaikan persoalan optimisasi dengan langkah dari simpul awal ke simpul jawaban terpendek maka algoritma BFS lebih cocok untuk digunakan dibandingkan algoritma DFS. Seperti yang dijelaskan pada pendahuluan, persoalan ini adalah persoalan optimisasi dan oleh karena itu algoritma BFS dipilih.

Algoritma BFS ini dapat digunakan untuk menyelesaikan banyak permasalahan asalkan permasalahan tersebut dapat dimodelkan dalam graf. Graf yang digunakan oleh algoritma ini bisa sudah ada sebelumnya (disebut sebagai representasi statis) ataupun dibuat saat penyelesaian pencarian (disebut sebagai representasi dinamis).

Cara kerja algoritma BFS secara lebih rinci adalah sebagai berikut:

1. Misalkan simpul awal adalah v, masukkan simpul tersebut ke dalam *queue* simpul hidup

2. Ambil elemen pertama pada *queue* simpul hidup menjadi simpul ekspan dan periksa apakah simpul ini merupakan solusi, jika bukan maka lanjutkan ke langkah 3. Apabila simpul ini merupakan solusi dan hanya memerlukan satu solusi maka pencarian selesai sampai di sini, jika memerlukan lebih dari satu solusi atau ingin mencari solusi terbaik maka lanjutkan ke langkah 3.

3. Bangkitkan seluruh simpul yang berhubungan dengan simpul ekspan ini yang belum pernah dikunjungi sebelumnya dan masukkan ke dalam *queue* simpul hidup.

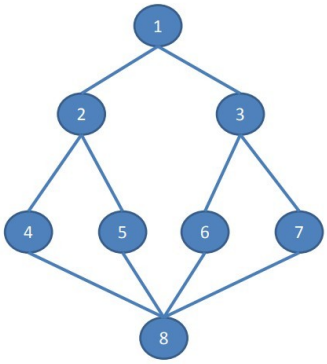
4. Ulangi langkah 2 dan 3 hingga *queue* simpul hidup kosong. Apabila saat *queue* simpul hidup kosong solusi masih belum ditemukan maka pencarian dinyatakan gagal.

```
BFS (G, s)                    //Where G is the graph and s is the
source node
      let Q be queue.
      Q.enqueue( s ) //Inserting s in queue until all its neighbour
vertices are marked.

      mark s as visited.
      while ( Q is not empty)
           //Removing that vertex from queue,whose neighbour will
be visited now
           v  =  Q.dequeue( )

          //processing all the neighbours of v
          for all neighbours w of v in Graph G
               if w is not visited
                         Q.enqueue( w )              //Stores w in
Q to further visit its neighbour
                         mark w as visited.
```

Gambar 2.2 Contoh Pseudocode Algoritma BFS
(Sumber : https://www.hackerearth.com/practice/algorithms/graphs/breadth-first-search/tutorial/)



Gambar 2.3 Contoh Graf Algoritma BFS

(Sumber : http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2019-2020/BFS-dan-DFS-(2020).pdf)

Iterasi	V	Q	dikunjungi							
			1	2	3	4	5	6	7	8
Inisialisasi	1	{1}	T	F	F	F	F	F	F	F
Iterasi 1	1	{2,3}	T	T	T	F	F	F	F	F
Iterasi 2	2	{3,4,5}	T	T	T	T	T	F	F	F
Iterasi 3	3	{4,5,6,7}	T	T	T	T	T	T	T	F
Iterasi 4	4	{5,6,7,8}	T	T	T	T	T	T	T	T
Iterasi 5	5	{6,7,8}	T	T	T	T	T	T	T	T
Iterasi 6	6	{7,8}	T	T	T	T	T	T	T	T
Iterasi 7	7	{8}	T	T	T	T	T	T	T	T
Iterasi 8	8	{}	T	T	T	T	T	T	T	T

Gambar 2.4 Tabel Penyelesaian dari Gambar 2.3
(Sumber : http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2019-2020/BFS-dan-DFS-(2020).pdf)

C. *Sdorica*

Sdorica merupakan sebuah *video game* dengan *genre fantasy strategy RPG* yang dapat dimainkan di Android dan IOS. Bagian utama permainan dari Sdorica adalah bertarung dengan tiga karakter, maksimal, dengan *skill* yang dimiliki masing-masing karakter unik dan berbeda dan diaktifkan menggunakan *command zone*, dengan cara mencocokkan *orb* sesuai warna posisi karakter, untuk melawan musuh-musuh.

Gambar 2.5 Contoh Tampilan Petarungan di Sdorica
(Sumber : Dokumentasi penulis)

Gambar di atas adalah contoh tampilan yang terlihat saat bertarung pada game Sdorica. Tiga karakter di kiri adalah karakter milik player. Sementara itu karakter di sebelah kanan adalah karakter musuh. Kumpulan 'bola api', yang disebut sebagai *orb* di dalam game, yang disusun dalam bentuk matriks 2x7 di bagian bawah gambar adalah *command zone*.

Karakter pada game Sdorica saat bertarung memiliki *Health*, *Armor*, dan *Status Effect*. Apabila karakter tersebut ada pada tim pemain maka karakter akan memiliki *Position* atau Posisi yang berbeda-beda. Sedangkan jika karakter berada pada tim lawan maka karkter tersebut akan memiliki *Turn Count*. *Turn Count* pada karakter musuh menandakan berapa giliran (*turn*) sebelum karakter musuh tersebut akan mengaktifkan salah satu *skill* miliknya.

Health menandakan berapa banyak *damage* yang dapat dikenakan ke karakter sebelum karakter tersebut kalah. Apabila *Health* mencapai nol maka karakter akan kalah dan hilang jika di tim musuh dan menjadi *dead* jika di tim pemain. *Health* ini adalah yang paling penting dijaga pada karakter pemain dan yang paling penting dihabiskan pada karakter lawan.

Armor adalah berapa banyak *damage* yang bisa dikenakan ke karakter sebelum *Health* dari karakter tersebut berkurang. Karater pada umumnya tidak memiliki *Armor* pada awal pertarungan tapi didapatkan dengan menggunakan salah satu *skill* karakter tersebut. Meskipun guna dari *Armor* adalah untuk menahan *damage* dari mengurangi *Health* suau karakter, ada beberapa *skill* yang memiliki *damage* yang langsung mengenai *Health* dan tidak mempedulikan apakah karakter tersebut masih memiliki *Armor* ataupun tidak.

Status Effect pada suatu karakter bisa mengubah cara kerja salah satu *skill* karakter tersebut atau bahkan mengubah cara kerja karakter tersebut secara keseluruhan. Pada umumnya ada dua jenis *Status Effect* yaitu *buff* dan *debuff*. Sebuah *buff* merupakan *Status Effect* yang menguntungkan bagi karakter yang mendapatkannya, misalnya menambahkan kekuatan *skill* atau memulihkan *Health* secara berkala. Sementara itu sebuah *debuff* merupakan *status effect* yang merugikan bagi karakter yang mendapatkannya, misalnya mengurangi kekuatan *skill* atau membuat karakter kehilangan *turn* miliknya dan tidak bisa melakukan apa-apa. Selain *buff* dan *debuff* ada juga *status effect* khusus yang hanya bisa diberikan oleh salah satu karakter dan hanya bisa digunakan oleh karakter itu sendiri dan umumnya digunakan untuk mengubah kemampuan karakter.

Setiap karakter memiliki *skill* yang unik bagi karakter tersebut. *Skill* dari karakter bisa saja memberikan *damage* yang mengurangi *Health* karakter yang terkena *skill* tersebut, memberikan *damage* dua kali lipat ke *Armor* karakter yang terkena *skill* tersebut atau lansung mengurangi *Health* tanpa mempedulikan *armor*. *Skill* dari suatu karakter juga bisa saja memberikan *buff* dan *debuff* kepada dirinya sendiri atau karakter tim pemain maupun lawan. Terkadang *Skill* dari suatu karakter juga dapat membuat karakter lain menggunakan *skill* nya.

Skill dari karakter pemain dibedakan menjadi 3 jenis, umumnya *1-orb skill*, *2-orb skill*, dan *4-orb skill. 1-orb skill* umumnya adalah *skill* paling lemah dan *4-orb skill* adalah *skill* paling kuat. Ada juga jenis-jenis *skill* lainnya seperti *3-orb skill* dan *6-orb skill* tapi keduanya lebih jarang dan biasanya menggantikan *2-orb skill* dan *4-orb skill*.

Sementara itu, *skill* yang digunakan oleh karakter musuh bermacam-macam dan pemain tidak dapat mengetahui *skill* apa saja yang dimiliki karakter musuh. Saat *turn count* karakter musuh mencapai angak nol (0) tidak dapat dipapstikan *skill* apa yang akan digunakan karakter musuh. Akan tetapi *skill* yang digunakan tidaklah acak dan pemain dapat memperhatikan keadaan karakter musuh dan pemain untuk menebak *skill* yang akan digunakan karakter musuh.

Posisi pada tim pemain dibagi menjadi tiga. Posisi *white* yang berada di paling kiri tim karakter pemain. Posisi *gold* yang berada di paling kanan tim karakter pemain. Serta Posisi

black yang berada di antara karakter pada position *white* dan *gold*.

Karakter yang berada pada Posisi *white* memiliki fungsi sebagai *support* atau *healer* dari tim. Karakter-karakter pada Posisi ini umumnya tidak memiliki *skill* yang banyak menyerang lawan secara langsung. Akan tetapi karakter-karakter pada Posisi ini umumnya memilik *skill* yang memberikan *buff* kepada karakter pemain dan *debuff* kepada musuh. Contohnya saja pada gambar tersebut karakter pada posisi *white* adalah Tica yang memiliki *1-orb skill* yang dapat menyembuhkan salah satu karakter pemain, *2-orb skill* yang dapat menyerang karakter musuh manapun, dan *4-orb skill* yang dapat menyembuhkan salah satu karakter pemain dengan *jauh* lebih baik dan *1-orb skill* miliknya dan juga meningkatkan pertahanan karakter tersebut.

Sementara itu, karakter yang ditengah adalah karakter pada posisi *black* dan memiliki fungsi sebagai *DPS* atau karakter penyerang. Karakter-karakter ini pada umumnya memiliki *skill* yang seluruhnya dapat menyerang musuh dan memiliki potensi *damage* yang paling tinggi diantara ketiga posisi karakter, tapi hampir sama sekali tidak memiliki *skill* lain selain untuk menyerang. Contohnya pada gambar tersebut karakter pada posisi *black* adalah Pang SP yang memiliki *1-orb skill* yang menyerang karakter lawan paling belakang (paling kanan di gambar) dan jika dengan serangan itu karakter lawan tersebut kehabisan *health* maka Pang SP akan menyerang karakter didepannya (jika ada) dengan *1-orb skill* miliknya lagi, 2-*orb skill* yang menyerang karakter lawan paling belakang dan jika dengan serangan itu karakter lawan tersebut kehabisan *health* maka Pang SP akan menyerang karakter didepannya (jika ada) dengan kekuatan yang sama dengan *1-orb skill* miliknya, dan *4-orb skill* yang menyerang karakter lawan paling belakang dan jika dengan serangan itu karakter lawan tersebut kehabisan *health* maka Pang SP akan menyerang karakter didepannya (jika) ada dengan kekuatan yang sama dengan *2-orb skill* miliknya.

Sementara itu, karakter paling depan adalah karakter dengan posisi *gold* yang memiliki fungsi umum sebagai *tanker* dari tim. Karakter-karakter ini umumnya memiliki sejumlah *skill* menyerang yang memiliki potensi *damage* lebih rendah dari *skill* yang dimiliki karakter pada posisi *black* dan juga satu *skill* yang dapat memulihkan *health* atau menaikkan pertahanan diri sendiri. Contohnya pada gambar tersebut karakter pada posisi *gold* adalah Law yang memiliki *1-orb skill* yang akan memulihkan sebagian *health* miliknya dan memberikan dirinya sendiri *status effect fury* jika ia tidak memiliki *status effect fury, 2-orb skill* yang akan menyerang karakter paling depan (paling dekat ke karakter ini dan di depannya) dari karakter tim lawan dengan serangan yang dua kali lebih efektif ke *armor* serta memberikan dirinya sendiri *status effect fury* jika ia hanya memiliki satu *status effect fury,* 4-*orb skill* yang akan menyerang karakter paling depan dari karakter tim lawan dengan serangan yang dua kali lebih efektif ke *armor* dan memberikan karakter tersebut *status effect tumble* serta memberikan dirinya sendiri *status effect fury* jika ia hanya memiliki dua *status effect fury*, dan *skill* yang otomatis aktif jika Law memiliki tiga *status effect fury* yang akan menyerang karakter manapun di tim lawan.

Command zone adalah bagian di sebelah bawah gambar yang memuat kumpulan 'bola api' bernama *orb* dari tiga warna dalam susunan matriks dua kali tujuh. *Command zone* inilah cara utama pemain memainkan Sdorica dan cara utama mengaktifkan *skill* dari karakter di tim pemain. Setiap kali pemain menggunakan *orb* yang ada pada *comman zone* maka akan muncul *orb* baru dari sisi kanan *command zone* yang akan menggantikan *orb* yang telah digunakan sehingga total *orb* pada *command zone* tetap membentuk matriks 2x7. Jenis (warna) *orb* yang baru ditentukan secara acak, tetapi jenis *orb* yang paling jarang digunakan pemain pada petarungan itu memiliki peluang yang lebih besar untuk muncul.

Orb pada *command zone* ada tiga warna, masing masing merepresentasikan posisi pada tim pemain. *Orb* berwarna putih merepresentasikan karakter pada posisi *white* dan digunakan untuk mengaktifkan *skill* yang dimiliki karakter pada posisi *white. Orb* berwarna ungu gelap merepresentasikan karakter pada posisi *black* dan digunakan untuk mengaktifkan *skill* yang dimiliki karakter pada posisi *black. Orb* berwarna kuning keemasan merepresentasikan karakter pada posisi *gold* dan digunakan untuk mengaktifkan *skill* yang dimiliki karakter pada posisi *gold.*

Skill yang diaktifkan bergantung dari jumlah dan warna *orb* yang digunakan. Apabila pemain menggunakan satu buah *orb* berwarna kuning maka *skill 1-orb* karakter pada posisi *gold* yang akan aktif, apabila pemain menggunakan dua buah *orb* berwarna ungu gelap secara lurus secara vertikal atau horizontal maka *skill* 2-*orb* karakter pada posisi *black* yang akan aktif, dan apabila pemain menggunakan empat buah *orb* berwarna putih sehingga membentuk sebuah kotak ukuran 2x2 maka *skill* 4-*orb* karakter pada posisi *white* yang akan aktif. Pengaktifan *skill* menggunakan *orb* ini tidak dapat dicampur-campur, pemain tidak dapat menggunakan *orb* dalam warna yang berbeda dalam satu *turn*, sehingga tidak mungkin menggunakan dua buah *orb* dengan satu berwarna hitam dan satu berwarna putih untuk mengaktifkan *skill 1-orb* dari karakter posisi *black* dan *white*. Akan tetapi apabila ingin menggunakan *skill* dari dua karakter sekaligus maka salah satu *skill* karakter tersebut harus dapat mengaktifkan *skill* karakter lainnya atau membuat musuh mengaktifkan *skill* karakter satunya lagi. Meskipun sebagian besar karakter memiliki konfigurasi *orb* untuk mengaktifkan *skill* yang terdiri dari *1-orb*, *2-orb* secara vertikal atau horizontal, dan *4-orb* yang membentuk kotak berukuran 2x2, ada beberapa karakter yang memiliki konfigurasi pengaktifan *skill* dalam bentuk *3-orb* dalam bentuk siku, *3-orb* secara horizontal, dan 6-*orb* dalam bentuk persegi panjang dengan ukuran 2x3.

Orb pada *command zone* tidak hanya digunakan untuk mengaktifkan *skill*. Apabila salah satu karakter pada tim pemain kehabisan *health* dan menjadi *dead* maka pemain dapat menghidupkan kembali (*resurrect*) karakter tersebut dengan menggunakan sejumlah *orb* dengan warna yang merepresentasikan posisi karakter tersebut. Jumlah *orb* yang diperlukan untuk menghidupkan kembali salah satu karakter bergantung pada karakter itu sendiri. Akan tetapi, aturan penggunaaan *orb* untuk menghidupkan kembali karakter sama dengan aturan menggunakan *orb* untuk mengaktifkan *skill* karakter. Yaitu, pada satu turn pemain hanya bisa menggunakan satu warna saja dan hanya bisa menggunakan

dalam konfigurasi satu *orb*, dua *orb* secara vertikal atau horizontal, atau empat orb yang membentuk sebuah kotak dengan ukuran 2x2. Apabila *skill* dari karakter diaktifkan dengan kofigurasi orb yang berbeda, misalnya *3-orb* atau *6-orb* maka konfigurasi *orb* yang digunakan untuk membangkitkan kembali karakter tersebut mengikuti konfigurasi pengaktifan *skill* karakter tersebut.

Setiap kali pemain menggunakan *orb* untuk mengaktifkan salah satu *skill* karakter pada timnya atau untuk menghidupkan kembali karakter pada timnya yang *dead* maka *turn* milik karakter akan berakhir dan *turn count* seluruh karakter lawan yang masih memiliki *health* di atas nol akan berkurang satu. Apabila ada karakter lawan yang *turn count* miliknya menjadi nol, maka karakter tersebut akan mengaktifkan salah satu *skill* miliknya dan mengubah *turn count* miliknya menjadi suatu angka yang lebih besar dari nol, angka tersebut bergantung kepada karakter itu sendiri serta keadaan (*health*, *armor*, *status effect*) karakter yang ada pada tim pemain dan pada tim lawan. Apabila tidak ada karakter lawan yang memiliki *turn count* benilai nol maka pemain mendapatkan *turn* kembali dan dapat menggunakan *orb* dan barang yang ia miliki kembali.

Pergantian *turn* ini terus terjadi hingga pemain kalah, yaitu seluruh karakter yang ada pada tim pemain menjadi *dead*, atau pemain menang, yaitu seluruh karakter pada tim musuh kehabisan *health*. Apabila pemain kalah maka pemain akan dikembalikan ke menu utama *game* dan harus memulai ulang misi tersebut dari awal. Apabila pemain berhasil mengalahkan tim lawan maka pemain akan dapat melanjutkan misinya, jika tim lawan tersebut bukan tim lawan terakhir pada misi tersebut, dan menjelajahi daerah pada misi tersebut serta menuju ke tim lawan berikutnya. Saat pemain bertemu tim lawan lain maka keadaan *health* setiap karakter pada tim pemain serta keadaan *command zone* akan sama dengan saat petarungan dengan tim lawan sebelumnya berakhir.

Karena *4-orb* sangat berguna untuk didapatkan supaya dapat menggunakan *skill* terkuat karakter dan menghidupkan kembali karakter yang sudah *dead* dalam jumlah *turn* sekecil mungkin maka penting untuk mengetahui cara mendapatkan *4-orb* setiap jenis *orb* dengan jumlah turn sesedikit mungkin.

III. Analisis Permasalahan

A. Pemodelan

Dalam bab-bab sebelumnya telah dijelaskan bahwa dalam permainan Sdorica medapatkan *4-orb* pada *command zone* dalam jumlah *turn* sesedikit mungkin itu penting, maka persoalan ini dapat dianggap sebagai persoalan optimisasi dan karenanya dapat digunakan algoritma BFS.

Dalam menyelesaikan persoalan ini keadaan dari *command zone* dimodelkan sebagai simpul, dengan *orb* digeser ke kiri seperti sebenarnya di dalam *game* tetapi tidak mengalami penciptaan *orb* baru dari kanan karena tidak mungkin mengetahui *orb* apa yang akan muncul. Sementara itu simpul-simpul merepresentasikan langkah-langkah yang perlu diambil pada setiap *turn* supaya bisa mencapai *goal,* yaitu sebuah keadaan *command zone* yang memiliki *4-orb*. Karena sebagian besar karakter memiliki *skill* yang diaktifkan dengan konfigurasi *orb 1-orb*, *2-orb*, dan *4-orb* berbentuk kotak maka ketiga konfigurasi tersebutlah yang dianggap valid untuk memudahkan pemodelan. Posisi dari *orb* yang perlu digunakan untuk mencapai keadaan *goal* pada setiap *turn* menyesuaikan dengan posisi *orb* setelah digeserkan ke kiri.

B. Implementasi

Berikut adalah algoritma BFS yang digunakan untuk menyelesaikan persoalan ini (dalam bahasa python3), dengan graf menggunakan modul anytree

```
def BFS(root_cz : cz.command_zone, goal_color :
cz.orb_type):
  # Defines and other preparations
  root = anytree.Node(root_cz.name, state=root_cz, lv=0)
  live_nodes = Q.SimpleQueue()
  live_nodes.put(root)
  expand_node = None
  goal_node = None
  goal_path = None
  goal_orb_count = 0

  # Checks if the goal (4-orb of the goal_color) is possible
  for i in range(0,
cz.command_zone_settings.ROW_SIZE.value):
      for j in range(0,
cz.command_zone_settings.COL_SIZE.value):
          if root_cz.orbs[i][j] == goal_color:
              goal_orb_count += 1

# The main BFS algorithm
  if (goal_orb_count > 3):
    while (not live_nodes.empty()):
      expand_node = live_nodes.get()
```

```
        # Checks if goal's reached
        for j in range(0,
cz.command_zone_settings.COL_SIZE.value - 1):
            if expand_node.state.orbs[0][j] == goal_color:
                if expand_node.state.orbs[0][j] ==
expand_node.state.orbs[1][j] and expand_node.state.orbs[0][j]
== expand_node.state.orbs[0][j+1] and
expand_node.state.orbs[1][j] == expand_node.state.orbs[1]
[j+1] :
                    neo_goal_path = getGoalPath(root,
expand_node)
                    if goal_path is None: # First solution found
                        goal_node = expand_node
                        goal_path = neo_goal_path
                    elif (len(neo_goal_path) < len(goal_path)): #
Checks if this is a better solution than the current one
                        goal_node = expand_node
                        goal_path = neo_goal_path

# Expands and such
        rems =
expand_node.state.find_possible_paths(goal_color)
        for rem in rems:
            temp_cz =
cz.command_zone(expand_node.state.orbs, rem)
            if (anytree.find(root, lambda node: node.name ==
temp_cz.name) is None): # If the node was already in the tree,
ignore it
                temp_node = anytree.Node(temp_cz.name,
parent=expand_node, state=temp_cz, lv=(expand_node.lv +
1))
                live_nodes.put(temp_node)

    if (goal_node != None): # A solution was found
        print("A 4-orb was found!")
        print(goal_node.state)
        printBFSSolution(goal_path)
    else: # No solution, failure
        print("4-orb cannot be created")
```

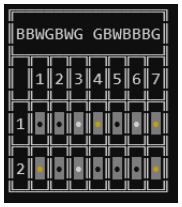
Gambar 3.1 Implementasi BFS untuk penyelesaian persoalan dalam Pyhton 3
(Sumber : Dokumentasi penulis)

C. Analisis Hasil

Keadaan *command zone* pada gambar 2.5 akan digunakan sebagai contoh karena *command zone* tersebut memiliki tiga keadaan yang cocok untuk melakukan tes, yaitu satu warna tidak dapat dibuat *4-orb*, satu warna dapat dibuat *4-orb* dengan mudah, dan satu warna lainnya dapat dibuat *4-orb* namun harus menghapus seluruh *orb* lainnya. Jika dimodelkan, keadaan *command zone* pada gambar 2.5 adalah sebagai berikut (dengan G untuk *orb* berwarna kuning keemasan, B untuk *orb* berwarna ungu gelap, dan W untuk *orb* berwarna putih) :

B B W G B W G
G B W B B B G

Masukan di atas dibaca oleh program seperti pada gambar 3.2 berikut ini. Titik berwarna kuning menandakan *orb* berwarna kuning, titik berwarna hitam menandakan *orb* berwarna ungu gelap, dan titik berwarna putih menandakan *orb* berwarna putih.

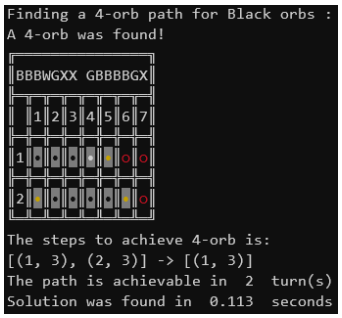


Gambar 3.2 Hasil pembacaan input program untu konfigurasi *command zone* pada gambar 2.5
(Sumber : Dokumentasi penulis)

Apabila yang diminta adalah konfigurasi *4-orb* untuk *orb* berwarna kuning program akan memberikan ouput seperti pada gambar 3.3 berikut ini. (Dengan titik berwarna merah menandakan posisi ‘kosong’ yang nantinya akan diisi *orb* berwarna acak oleh game)

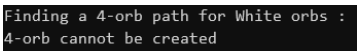
Gambar 3.3 Hasil keluaran program dari input pada gambar 3.2 untuk *orb* berwarna kuning
(Sumber : Dokumentasi penulis)

Apabila yang diminta adalah konfigurasi *4-orb* untuk *orb* berwarna ungu gelap program akan memberikan ouput seperti pada gambar 3.4 berikut ini.



Gambar 3.4 Hasil keluaran program dari input pada gambar 3.2 untuk *orb* berwarna ungu gelap
(Sumber : Dokumentasi penulis)

Apabila yang diminta adalah konfigurasi *4-orb* untuk *orb* berwarna putih program akan memberikan ouput seperti pada gambar 3.5 berikut ini.



Gambar 3.5 Hasil keluaran program dari input pada gambar 3.2 untuk *orb* berwarna putih
(Sumber : Dokumentasi penulis)

Dari hasil di atas dapat dilihat bahwa algoritma BFS cukup efektif dalam menemukan langkah-langkah terpendek untuk membuat konfigurasi *4-orb* dari keadaan suatu *command zone* pada *game* Sdorica. Keadaan *orb* warna kuning adalah salah satu *wors case* dari persoalan ini karena harus mengekspan banyak sekali simpul karena keadaan *command zone* yang hanya tepat memiliki empat *orb* berwarna kuning dan semuanya berada di ujung-ujung *command zone*. Meskipun begitu algoritma berhasil menyelesaikan masalah tersebut dengan cukup cepat yaitu 2,624 detik.

Keadaan *orb* berwarna ungu gelap adalah keadaan yang cukup baik karena memiliki cukup banyak *orb* berwarna tersebut dan cukup tersebar di dalam *command zone* sehingga tidak perlu membangkitkan banyak simpul. Waktu penyelesaiannya pun cepat dengan hanya 0,113 detik.

Sementara itu keadaan *orb* berwarna putih adalah keadaan gagal karena dengan keadaan *command zone* seperti pada gambar 2.5 tidak mungkin membuat konfigurasi *4-orb* berwarna putih.

Dari ketiga hasil keluaran tersebut terlihat bahwa algorimta BFS ini bisa menemukan langkah terpendek untuk membentuk konfigurasi *4-orb* pada game Sdorica dengan cukup baik. Untuk sebuah kasus yang mendekati kasus terbaik algoritma berhasil menemukan solusi yang hanya memerlukan dua *turn*. Sementara itu untuk salah satu kasus terburuk sekalipun algoritma berhasil menemukan solusi yang hanya memerlukan enam *turn* padahal jumlah *orb* yang perlu digunakan sebelum bisa mendapatkan konfigurasi *4-orb* tersebut adalah sepuluh *orb*.

Meskipun begitu kecepatan algoritma ini masih kurang bagus, terlebih jika seandainya ukuran *command zone* diperbesar. Ada baiknya jika jumlah simpul yang dibangkitkan dibuat lebih sedikit, atau sebagian perbandingan dikurangi supaya meningkatakan performa.

IV. Simpulan

Seperti yang ditunjukkan pada bagian sebelumnya, algotima BFS dapat dengan baik menemukan langkah terpendek untuk membuat konfigurasi *4-orb* dari setiap jenis *orb* dari *command zone* yang diberikan dalam waktu yang cukup baik. Meskipun begitu pemodelan yang digunakan pada makalah ini tidak dapat bekerja jika konfigurasi pengaktifan *skill* dari karakter pada tim pemain ada yang bukan *1-orb*, *2-orb* horizontal atau vertikal, dan *4-orb* berbentuk kotak 2x2. Selain itu pemodelan ini tidak dapat memprediksi *orb* yang akan diberikan oleh *game*, bisa saja kasus terbaik pada algoritma ini justru lebih menguntungkan karena dengan satu *turn* langsung diberikan *orb* yang sesuai untuk membuat konfigurasi *4-orb* di ujung kanan *command zone*. Terakhir, apabila ingin menggunaan pemodelan pada makalah ini ada baiknya performanya ditingkatkan, baik itu dengan mengurangi jumlah simpul yang dibangkitkan atau dengan mengurangi perbandingan yang dilakukan.

Video Link at Youtube

https://youtu.be/xLQu4fNZnVI

Acknowledgment

Puji dan syukur penulis ucapkan kepada Allah SWT karena tanpa rahmat dan karunianya penulis tidak akan dapat menyelesaikan makalah yang berjudul "Penerapan Algoritma BFS untuk Menentukan Langkah Terpendek Mendapatkan 4-orb dalam Permainan Sdorica" ini.

Penulis juga berterima kasih kepada tim dosen pengajar mata kuliah Strategi Algoritma IF2211 ini karena telah memberikan ilmu-ilmu yang memungkinkan makalah ini terbentuk.

Terakhir penulis mengucapkan terima kasih kepada rekan dan kerabat yang telah membantu penulis menyelesaikan makalah ini.

Referensi

[1] http://informatika.stei.itb.ac.id/~rinaldi.munir/Matdis/2015-2016/Graf (2015).pdf terakhir diakses 4 Mei 2020

[2] http://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2019-2020/BFS-dan-DFS-(2020).pdf terakhir diakses 4 Mei 2020

[3] https://web.archive.org/web/20140828203855/http://www.pong-story.com/intro.htm terakhir diakses 2 Mei 2020

[4] https://www.britannica.com/topic/electronic-strategy-game terakhir diakses 2 Mei 2020

[5] https://www.britannica.com/topic/role-playing-video-game terakhir diakses 2 Mei 2020

[6] https://www.sdorica.com/en/ terakhir diakses 2 Mei 2020

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, 4 Mei 2020

Hanif Muhamad Gana 13518127