Penggunaan BFS, DFS, dan UCS Untuk Eksplorasi dalam Permainan Dungeon Crawler Pokémon Mystery Dungeons: Explorers of Sky

Maria Flora Renata Siringoringo - 13522010
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13522010@std.stei.itb.ac.id

***Abstract*—Makalah ini mendemonstrasi penggunaan algoritma penentuan rute dalam permainan *dungeon crawler* seperti Pokémon Mystery Dungeons: Explorers of Sky, di mana pemain harus menelusuri sebuah *dungeon* yang menyerupai labirin. Dengan merepresentasikan peta dalam permainan tersebut sebagai graf, kita dapat menelusurinya menggunakan tiga algoritma, yaitu BFS, DFS, dan UCS. Untuk seorang pemain biasa, algoritma DFS merupakan algoritma yang paling efektif dalam penelusuran *dungeon*, tetapi BFS dan UCS dapat menghasilkan rute yang lebih optimal jika rute dan peta dapat diproses tanpa menggerakan karakter pemain terlebih dahulu.**

Keywords—pathfinding; uninformed search; dungeon crawler

I. Pendahuluan

Dungeon crawler/crawling adalah sebuah skenario yang sangat umum digunakan dalam permainan video. Dalam sebuah *dungeon crawler*, pemain akan menelusuri sebuah lingkungan (*dungeon*) yang menyerupai labirin untuk mencapai sebuah obyektif seperti mengalahkan monster atau keluar dari lingkungan tersebut. Pada umumnya, pemain tidak diberikan peta dari *dungeon* tersebut, sehingga pemain harus menelusuri berbagai ruangan hingga menemukan jalan yang tepat.

Ada beberapa faktor yang mempersulit penelusuran sebuah *dungeon*, diantaranya adalah banyaknya karakter musuh yang dapat muncul di dalam *dungeon* tersebut, adanya jebakan, dan adanya kejadian-kejadian acak yang dapat merugikan. Selain itu, tata letak sebuah *dungeon* umumnya dibangkitkan secara acak setiap kali pemain mengunjunginya. Faktor-faktor tersebut mendorong seorang pemain untuk mencapai akhir dari *dungeon* dengan lebih cepat.

Pencarian jalan keluar dalam sebuah *dungeon* dapat dikategorikan sebagai sebuah masalah penentuan rute. Oleh karena itu, kita dapat menggunakan algoritma-algoritma pencarian rute untuk menyelesaikan permainan *dungeon crawler*. Makalah ini akan mendemonstrasikan penggunaan algoritma pencarian buta dalam menyelesaikan penelusuran *dungeon* pada permainan Pokémon Mystery Dungeons: Explorers of Sky.

II. Penentuan Rute

Penentuan rute adalah pencarian sebuah rute dari satu titik ke titik yang lain di antara beberapa pilihan jalan. Selain mencari rute yang dapat dilewati, penentuan rute biasanya juga mencari rute yang paling singkat antara kedua titik tersebut. Secara general, penentuan rute dapat dibagi menjadi dua jenis berdasarkan visibilitas lingkungan yang akan ditelusuri, yaitu *uninformed search* dan *informed search*.

Uninformed search adalah penentuan rute tanpa mengetahui lokasi tujuan dan seluruh jalan yang dapat dilalui sebelum sampai ke titik yang berhubungan dengannya. Karena informasi yang terbatas, *uninformed search* akan mencari ke berbagai arah hingga menemukan titik tujuan. Beberapa contoh algoritma *uninformed search* adalah breadth-first search, depth-first search, dan uniform cost search.

Sebaliknya, *informed search* dapat dilakukan jika lokasi tujuan dan seluruh titik diketahui. *Informed search* menggunakan sebuah fungsi heuristik untuk menentukan titik yang akan dikunjungi berikutnya. Dengan ini, pergerakan pada *informed search* biasanya lebih terarah ke tujuan dan akan mengunjungi lebih sedikit titik dibandingkan dengan *uninformed search*. Beberapa contoh algoritma *informed search* adalah greedy best-first search dan A*.

III. Dungeon Crawling

Dungeon crawling sebagai sebuah skenario permainan muncul pertama kali pada tahun 1975 pada artikel “Solo Dungeon Adventures” dalam majalah The Strategic Review edisi pertama. Artikel tersebut ditulis oleh Gary Gygax tentang sistem untuk bermain Dungeons and Dragons, sebuah permainan *RPG tabletop*, sendirian. Permainan video pertama yang memiliki konsep *dungeon crawling* adalah pedit5 yang dibuat pada tahun 1975 oleh Rusty Rutherford. Permainan tersebut dibuat untuk sistem PLATO yang berada di University of Illinois. PLATO adalah sistem bersama yang, pada saat itu, dapat diakses oleh 150 terminal yang berbeda. Hal tersebut berarti sumber daya yang dimiliki PLATO sangat terbatas, sehingga pedit5 dihapus oleh administrator sistem. Sistem karakter dalam permainan pedit5 terinspirasi oleh Dungeons

and Dragons, sedangkan permainannya sendiri meliputi eksplorasi sebuah *dungeon* yang tetap dengan monster yang dibangkitkan secara acak.

Selanjutnya, pada tahun 1976, Gary Whisenhunt dan Ray Wood merilis permainan dnd di sistem PLATO. Mereka sempat memainkan pedit5 dan mengembangkan permainan tersebut menjadi dnd. Kedua permainan tersebut kemudian menginspirasi permainan-permainan *dungeon crawler* lain di sistem PLATO.

Dungeon Crawler pertama di sistem komputer personal (PC) adalah Beneath Apple Manor pada 1978. Permainan tersebut dibuat untuk komputer Apple II. Perbedaan terbesar dari permainan video sebelumnya dengan Beneath Apple Manor adalah penggunaan pembangkit angka semi-acak untuk membuat *dungeon*, sehingga pemain akan memasuki *dungeon* yang berbeda setiap kali dia memainkan Beneath Apple Manor.

Pokémon Mystery Dungeons: Explorers of Sky (selanjutnya disebut sebagai Explorers of Sky) adalah sebuah permainan dalam seri *dungeon crawler* Pokémon Mystery Dungeons. Explorers of Sky dirilis pada tahun 2009 untuk Nintendo DS. Permainan tersebut dibuat oleh Chunsoft. Di dalam Explorers of Sky, pemain memainkan seekor Pokémon yang menjadi anggota dari sebuah serikat penjelajah. Karakter pemain akan menjelajahi *dungeon* yang dibangkitkan secara acak. Permainan tersebut memiliki 71 *dungeon*, masing-masing memiliki jumlah lantai, jenis Pokémon, dan jenis barang yang dapat ditemukan yang berbeda.

IV. Batasan Dan Pemilihan Algoritma Penentuan Rute

Eksplorasi dalam Pokémon Mystery Dungeons: Explorers of Sky selalu dimulai dari sebuah ruangan yang tersambung dengan satu atau lebih koridor. Sebuah koridor dapat terhubung dengan sebuah ruangan, koridor lain, atau tidak tersambung dengan apapun (jalan buntu). Jika direpresentasikan sebagai graf, ruangan, percabangan antara dua koridor, dan akhir dari jalan buntu merupakan simpul, sedangakan bagian koridor yang tidak bercabang adalah sisi.

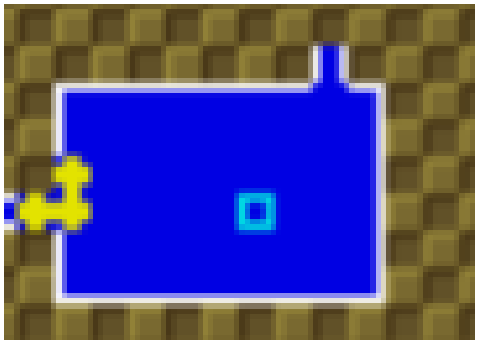
Peta di Explorers of Sky akan kosong pada awal eksplorasi. Peta akan muncul jika karakter pemain sudah cukup dekat dengan area yang belum ada pada peta. Oleh karena itu, metode penentuan rute yang dapat digunakan seorang pemain hanya *uninformed search*.

Gambar 1: Peta pada saat baru memasuki dungeon (Sumber: arsip penulis)

Makalah ini akan mencoba tiga algoritma, yaitu BFS, DFS, dan UCS. Alasan pemilihan ketiga algoritma tersebut adalah kesederhanaannya yang dapat diikuti oleh seorang pemain tanpa menggunakan bot khusus.

V. Aplikasi Algoritma

Ketiga algoritma akan memiliki simpul awal dan akhir yang sama. Simpul awal adalah ruangan di mana karakter pemain muncul pertama kali, sedangkan simpul akhir adalah ruangan yang memiliki tangga.



Gambar 2: Tampilan ruangan dengan tangga pada peta (Sumber: arsip penulis)

A. *Breadth-First Search*

Algoritma Breadth-First Search dapat diimplementasikan dengan sebuah *queue*. Algoritma ini akan mengunjungi semua tetangga sebuah simpul secara berurutan, kemudian semua tetangga dari tetangga simpul pertama, dan seterusnya. Algoritma akan berhenti jika *queue* kosong atau jika sudah sampai ke simpul tujuan. Dalam konteks Explorers of Sky, pseudocode algoritma Breadth-First Search sebagai berikut:

```
function BFS(RuanganAwal: Node)
  createQueue(Q)
  visited[RuanganAwal] <- True
  push(Q, RuanganAwal)
  found <- False
  while (not found and not empty(Q))
    pop(Q, n)
    visited[n] <- True
    if ada tangga di n then
      found <- true
    else {tidak ada tangga di n}
      for setiap v dalam getNeighbours(n) do
        if not visited[v] then
          push(Q, v)
```

Psuedocode di atas mengasumsikan bahwa ada struktur data Node dengan fungsi getNeighbours(n) untuk mendapatkan sebuah larik berisi semua ruangan, percabangan koridor, dan jalan buntu yang terhubung dengan n, dan juga struktur data Queue yang memiliki fungsi push(q, n) untuk menambahkan Node n ke posisi paling belakang Queue q, serta pop(q, n) untuk menghapus Node pada posisi paling depan q dan menaruh nilainya di n. Visited[] adalah sebuah Map yang memiliki kunci bertipe Node dan nilai bertipe boolean. Visited[n] bernilai True jika n sudah pernah dikunjungi, sedangkan False jika belum.

Kompleksitas waktu dari Breadth-First Search adalah $O(b^d)$, dengan b adalah jumlah pencabangan terbanyak dari sebuah simpul pada graf yang ditelusuri dan d adalah kedalaman dari rute optimal.

B. Depth-First Search

Algoritma Depth-First Search dapat diimplementasikan secara rekursif. Algoritma ini akan mengunjungi tetangga pertama sebuah simpul hingga sampai ke sebuah simpul yang semua tetangganya sudah dikunjungi atau tidak memiliki tetangga. Kemudian, algoritma akan kembali ke simpul terakhir yang sudah dikunjungi dan memiliki tetangga yang belum dikunjungi, lalu mengunjungi tetangga tersebut. Algoritma akan berhenti jika sudah sampai ke simpul tujuan atau jika semua simpul sudah dikunjungi. Dalam konteks Explorers of Sky, pseudocode algoritma Depth-First Search sebagai berikut:

```
function DFS(RuanganAwal: Node)
    visited[RuanganAwal] <- True
    if ada tangga di RuanganAwal then
        {return hasil}
    else {tidak ada tangga di RuanganAwal}
        for setiap v dalam getNeighbours(RuanganAwal) do
            if not visited[v] then
                return DFS(v)
    return {return hasil jika tidak ditemukan ruangan dengan tangga sama sekali}
```

Psuedocode di atas mengasumsikan bahwa ada struktur data Node dengan fungsi getNeighbours(n) untuk mendapatkan sebuah larik berisi semua ruangan, percabangan koridor, dan jalan buntu yang terhubung dengan n. Visited[] adalah sebuah Map yang memiliki kunci bertipe Node dan nilai bertipe boolean. Visited[n] bernilai True jika n sudah pernah dikunjungi, sedangkan False jika belum.

Kompleksitas waktu dari Depth-First Search adalah $O(b^m)$, dengan b adalah jumlah pencabangan terbanyak dari sebuah simpul pada graf yang ditelusuri dan m adalah kedalaman maksimum pohon status (sebuah pohon yang merepresentasikan semua simpul yang sudah dibangkitkan/menjadi simpul ekspan pada DFS dan BFS untuk graf dinamis).

C. Uniform Cost Search

Algoritma Uniform Cost Search sedikit berbeda dengan kedua algoritma sebelumnya. Uniform Cost Search mengukur jarak antara simpul awal dengan simpul berikutnya untuk menentukan simpul yang dikunjungi selanjutnya. Dengan perhitungan tersebut, Uniform Cost Search dapat menemukan rute yang paling pendek dari simpul awal ke simpul tujuan.

Pergerakan dalam Explorers of Sky bergantung pada petak pada lantai. Satu karakter atau benda berdiri dalam satu petak dan dapat bergerak satu petak setiap putaran, dengan delapan pilihan arah pergerakan sesuai mata angin (kanan, kiri, atas, bawah, dan secara diagonal).

Petak lantai di dalam sebuah *dungeon* terdiri dari beberapa tipe medan. Beberapa medan hanya dapat dilewati oleh Pokémon tertentu. Petak-petak dengan jenis tersebut akan dianggap tidak dapat dilewati agar solusi rute inklusif untuk setiap Pokémon yang dapat dimainkan oleh pemain.

Karena sistem peta pada Explorers of Sky tidak menunjukkan panjang dari sebuah koridor yang belum pernah dilewati pemain, pengukuran jarak antara ruangan awal dengan simpul-simpul lain harus diukur dari jumlah petak antara tengah ruangan dengan permulaan koridor (untuk simpul berupa ruangan) dan perkiraan untuk simpul berupa percabangan koridor. Oleh karena itu, Uniform Cost Search untuk penentuan rute pada Explorers of Sky kurang akurat, tapi akan mendekati.

Uniform Cost Search dapat diimplementasikan dengan sebuah *priority queue*, dengan nilai prioritas diisi jarak antara simpul awal dan simpul tujuan. Dalam konteks Explorers of Sky, pseudocode algoritma Uniform Cost Search sebagai berikut:

```
function UCS(RuanganAwal: Node)
    createPriorityQueue(Q)
    visited[RuanganAwal] <- True
    push_priority(Q, RuanganAwal, 0)
    found <- False
    while (not found and not empty(Q))
        pop(Q, n)
        visited[n] <- True
        if ada tangga di n then
            found <- true
        else {tidak ada tangga di n}
            for each getNeighbours(n) do
                if not visited[v] then
                    priority <- distance(RuanganAwal, v)
```

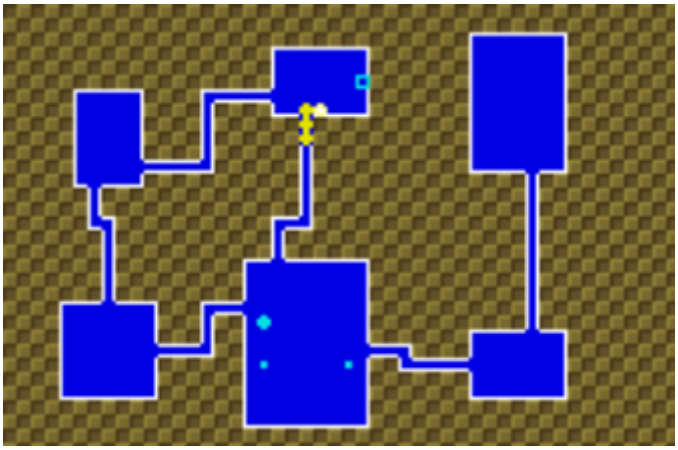
push_priority(Q, v, priority)

Psuedocode di atas mengasumsikan bahwa ada struktur data Node dengan fungsi getNeighbours(n) untuk mendapatkan sebuah larik berisi semua ruangan, percabangan koridor, dan jalan buntu yang terhubung dengan n, dan juga struktur data Priority Queue yang memiliki fungsi push_priority(q, n, p) untuk menambahkan Node n ke posisi yang sesuai dengan prioritas p, dengan prioritas yang terkecil berada di paling depan Priority Queue q, serta pop(q, n) untuk menghapus Node pada posisi paling depan q dan menaruh nilainya di n. Terdapat juga sebuah fungsi distance(n) yang akan mengembalikan jarak antara ruangan awal dengan simpul n.

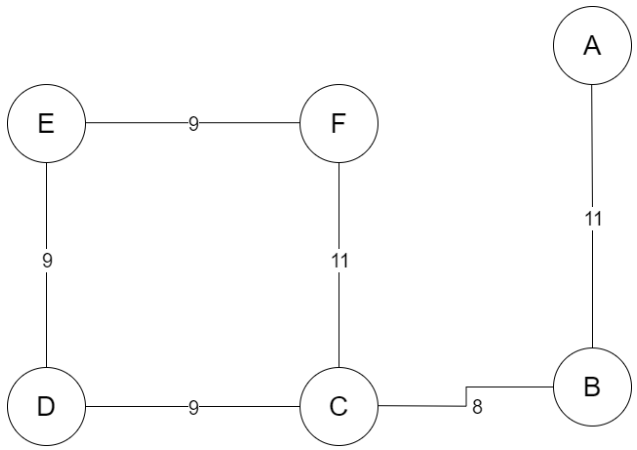
VI. PERCOBAAN

Pengujian akan dilakukan dengan emulator agar dapat dilakukan simpan dan muat *state* permainan, sehingga lantai *dungeon* yang sama dapat digunakan untuk 3 algoritma yang berbeda.

Dalam pengujian, karakter pemain akan digerakkan secara langsung (tanpa *bot*) dengan graf sesuai dengan peta *dungeon* pada setiap pergerakan. Pengujian akan mencatat urutan simpul yang diekspan/dikunjungi, jumlah petak yang dilalui pada koridor selama pencarian berlangsung (Langkah saat melakukan *backtrack* dihitung), jumlah simpul yang dikunjungi, dan rute akhir yang terbentuk oleh algoritma penentuan rute.



Gambar 3: Peta lantai yang digunakan untuk percobaan 1 (Sumber: arsip penulis)

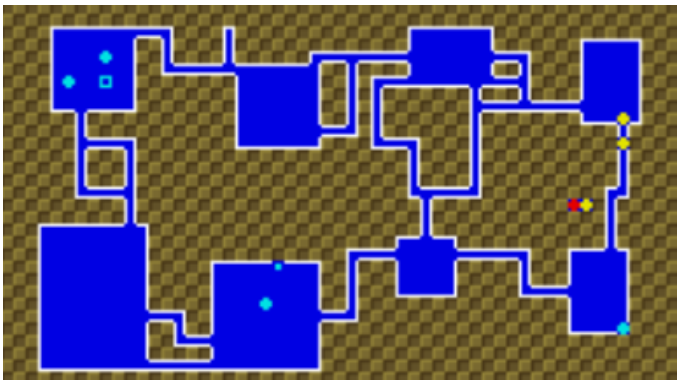


Gambar 4: Graf dari lantai yang digunakan untuk percobaan 1 (Sumber: arsip penulis)

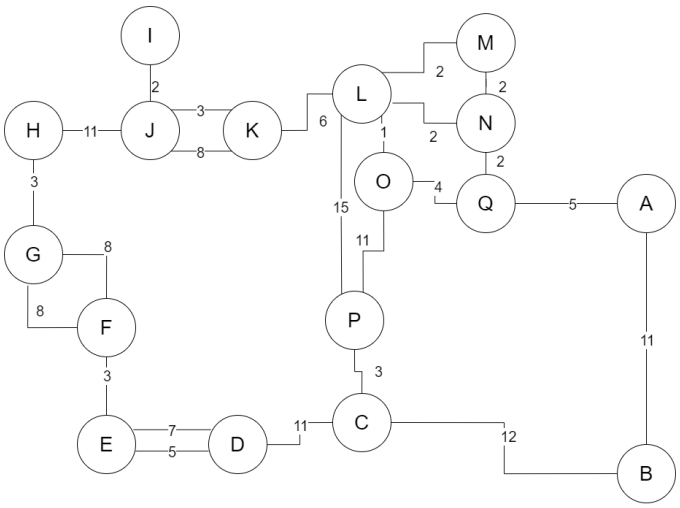
Simpul awal adalah A dan simpul tujuan adalah F. Hasil dari ketiga algoritma adalah sebagai berikut:

TABEL 1: PERCOBAAN 1

Algoritma	**Data**			
	Urutan Simpul Ekspan	***Jumlah petak dilalui selama pencarian***	***Simpul dikunjungi***	***Rute yang ditemukan***
BFS	ABCDF	48	5	ABCF
DFS	ABCDEF	51	6	ABCDEF
UCS	ABCDF	48	5	ABCF



Gambar 5: Peta lantai yang digunakan untuk percobaan 2 (Sumber: arsip penulis)



Gambar 6: Graf dari lantai yang digunakan untuk percobaan 2 (Sumber: arsip penulis)

Simpul awal adalah A dan simpul tujuan adalah H. Hasil dari ketiga algoritma adalah sebagai berikut:

TABEL 2: PERCOBAAN 2

Algoritma	**Data**			
	Urutan Simpul Ekspan	***Jumlah petak dilalui selama pencarian***	***Simpul dikunjungi***	***Rute yang ditemukan***
BFS	ABQCNODPLMEKFJGH	571	16	AQNLKJH
DFS	ABCDEFGH	53	8	ABCDEFG
UCS	AQNLMOBKJIPCH	131	13	AQNLKJH

Pada percobaan pertama, tata letak *dungeon* sangat sederhana dan hanya terdapat satu percabangan, sehingga hasil menggunakan BFS, DFS, dan UCS tidak terlalu berbeda. Algoritma BFS dan UCS melewati jumlah petak dan mengunjungi simpul yang sama, sedangkan DFS mengunjungi satu simpul lebih banyak dari kedua algoritma tersebut.

Pada percobaan kedua, terdapat perbedaan yang sangat besar antara ketiga algoritma. BFS melewati paling banyak petak dan mengunjungi paling banyak simpul sebelum berhasil menemukan simpul tujuannya., sedangkan UCS melewati 131 petak dan DFS hanya melewati 53 petak. Hal ini terjadi karena BFS dan UCS melakukan jauh lebih banyak *backtracking*.

Jika algoritma tersebut dijalankan secara langsung oleh seorang pemain, yang artinya menjalankan karakternya sesuai urutan penelusuran algoritma-algoritma tersebut, BFS menjadi sangat tidak efisien. Jika perlu menggerakan karakter secara langsung, *backtracking* akan membuat harga penelusuran sebuah jalan menjadi dua kali lebih besar, karena pemain harus berjalan kembali ke ruangan atau percabangan sebelumnya. Selain menambah waktu eksplorasi, *backtracking* juga tidak menghasilkan informasi baru pada peta, karena pemain hanya melewati jalan yang sudah dilewati sebelumnya.

Bertolak belakang dengan BFS, DFS jarang melakukan *backtracking*. DFS hanya melakukan *backtrack* jika terpaksa, yaitu jika sudah berada pada simpul yang tidak memiliki tetangga yang dapat dikunjungi. Karena hal tersebut, DFS dapat menelusuri ruangan dengan lebih cepat dan menghasilkan lebih banyak informasi pada peta.

UCS dapat bekerja menyerupai BFS dan DFS sesuai dengan persebaran harga sisi pada graf. Pada percobaan pertama, UCS bekerja seperti BFS karena harga setiap sisi pada graf memiliki perbedaan yang kecil. Jangkauan nilai sisi pada percobaan pertama adalah 3 petak. Sedangkan pada percobaan kedua, jangkauan nilai sisi adalah 13, dengan sisi paling kecil bernilai 2 dan sisi paling panjang bernilai 15. Karena itu, penelusuran menggunakan UCS dapat mengunjungi beberapa simpul tanpa melakukan *backtracking*, tidak seperti BFS yang melakukan *backtracking* setelah setiap simpul. Karena UCS dengan nilai seluruh sisi yang sama akan menjadi sama dengan DFS, maka jumlah petak dilewati yang paling banyak yang diperlukan oleh UCS adalah sebanyak pada DFS, sedangkan dalam kasus terbaik, UCS bisa melalui lebih sedikit petak dibandingkan dengan DFS.

Akan tetapi, UCS sangat sulit dilakukan oleh seorang pemain dalam Explorers of Sky. Hal ini disebabkan oleh peta yang baru terlihat jika pemain sudah dekat dengan sebuah petak, sehingga peta dari Explorers of Sky dapat dianggap sebagai graf dinamis. Dalam pelaksanaannya, pemain dapat melihat sedikit lebih jauh dibandingkan kemunculan sebuah petak pada peta, sehingga UCS masih dapat dilakukan sebagian dengan akurat. Jika semua koridor/sisi panjang, pemain tidak dapat memperkirakan panjang dari sisi tersebut, sehingga beberapa langkah UCS harus dilakukan dengan menebak. Hal tersebut sangat tidak efisien dan membuat UCS tidak cocok digunakan untuk masalah ini.

Maka, jika seorang pemain akan melakukan penentuan rute secara manual dengan menggerakan karakternya secara langsung, algoritma DFS adalah yang paling cocok untuk digunakan.

Akan tetapi, rute akhir yang ditemukan oleh DFS selalu lebih panjang dari rute yang ditemukan oleh BFS dan UCS. Pada kedua percobaan, rute yang ditemukan BFS dan UCS sama dan optimal. Hal ini terjadi karena DFS tidak memperhitungkan jarak antara simpul yang dikunjungi dengan simpul awal. Sesuai namanya, DFS melakukan pencarian secara mendalam, sehingga solusi yang ditemukan bisa saja lebih dalam dari solusi lain.

BFS melakukan pencarian secara melebar dan mengunjungi semua simpul pada satu tingkat kedalaman sebelum memulai pencarian pada tingkat kedalaman berikutnya. Hal ini menyebabkan banyaknya terjadi *backtracking*, tetapi rute yang ditemukan pasti berada pada tingkat kedalaman yang paling rendah/kecil. Hal ini mirip dengan UCS yang menentukan simpul yang akan dikunjungi hanya berdasarkan jarak simpul tersebut dengan simpul awal. Karena simpul-simpul yang lebih dekat dengan simpul awal didatangi terlebih dahulu, solusi pertama yang terbentuk pasti merupakan solusi optimal dengan

rute yang paling pendek jika dihitung jaraknya dari simpul awal.

Maka, jika penelusuran simpul (dan peta) dapat dilakukan tanpa menggerakan karakter pemain di dalam permainan, BFS dan UCS akan optimal. Hal tersebut dapat dilakukan oleh sebuah *bot* yang memiliki akses ke peta yang lengkap dan dapat menjalankan algoritma BFS atau UCS sebelum menggerakan karakter pemain sesuai dengan rute yang dihasilkan. Walaupun begitu, jika pemain atau *bot* memiliki akses ke peta lengkap, mereka juga dapat menggunakan algoritma *informed search* yang mungkin lebih cocok dan efisien (BFS dan UCS akan tetap optimal).

VII. Kesimpulan Dan Saran

Algoritma penentuan rute dapat digunakan oleh seorang pemain permainan *dungeon crawler* untuk menentukan urutan pencarian atau eksplorasi dalam sebuah *dungeon*. Algoritma yang dapat digunakan pada *dungeon crawler* secara umum adalah algoritma *uninformed search*. Untuk permainan Pokémon Mystery Dungeons: Explorers of Sky dan permainan serupa seperti permainan dalam seri Mystery Dungeons lain, algoritma yang paling cocok digunakan adalah Depth-First Search yang meminimalisir *backtracking*. Keunggulan lain dari Depth-First Search pada eksplorasi *dungeon* adalah simpul tujuan pasti ada di dalam graf, sehingga Depth-First Search tidak dapat melakukan *infinite loop*, tidak seperti DFS pada permasalahan lain.

Ucapan Terima Kasih

Penulis berterimakasih kepada Ibu Nur Ulfa Maulidevi selaku dosen pengajar kelas Strategi Algoritma penulis. Penulis juga berterimakasih kepada teman-teman dan adiknya yang memberi semangat kepada penulis.

Referensi

[1] Munir Rinaldi, Maulidevi Nur Ulfa. “Breadth/Depth First Search (BFS/DFS) (Bagian 2)”. *informatika.stei.itb*, Rinaldi Munir, 2021. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag2.pdf. Accessed 11 June 2024.

[2] Rinaldi Munir, Nur Ulfa Maulidevi. “Penentuan Rute (Route/Path Planning Bagian 1: BFS, DFS, UCS, Greedy Best First Search”. *informatika.stei.itb*, Rinaldi Munir, 2021. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/Route-Planning-Bagian1-2021.pdf. Accessed 11 June 2024.

[3] Kai Li Lim, Kah Phooi Seng, Lee Seng Yeong, Li-Minn Ang, Sue Inn Ch’ng. “Uninformed pathfinding: A new approach”. in Expert Systems with Applications, vol. 42, Issue 5, 2015, pp. 2722-2730.

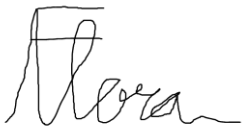
[4] Barnouti N. H., Al-Dabbagh S. S. M., Naser M. A. S.. "Pathfinding in Strategy Games and Maze Solving Using A* Search Algorithm". in Journal of Computer and Communications, vol.4 no.11, 2016.

[5] Brewer, Nathan. “Going Rogue: A Brief History of the Computerized Dungeon Crawl.” *IEEE-USA InSight*, IEEE, 7 July 2016, insight.ieeeusa.org/articles/going-rogue-a-brief-history-of-the-computerized-dungeon-crawl/. Accessed 12 June 2024.

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, 12 Juni 2024



Maria Flora Renata S - 13522010