Implementasi dan Analisis Algoritma untuk Menyelesaikan Permasalahan *Maze*

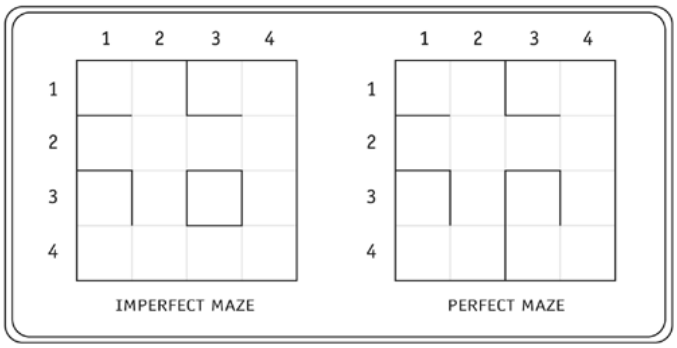
WAHYU PRIHANTORO – 1406579100
Universitas Indonesia
Fakultas Ilmu Komputer
Kampus UI Depok, Indonesia 16424

***Abstrak*—Paper ini akan membahas tentang permasalahan *maze* dengan beberapa implementasi dan analisis algoritma penyelesaian di antaranya adalah *Wall Follower Algorithm*, *Depth-First Search Algorithm, Breadth-First Search Algorithm*. Dari segi *running time* algoritma *Wall-Follower* merupakan algoritma terbaik. Namun jika dilihat dari hasil *path* yang ditemukan, algoritma *Breadh-First Search* terjamin menghasilkan *path* terpendek.**

Keywords—bfs; dfs; maze; vertex;

I. PENDAHULUAN

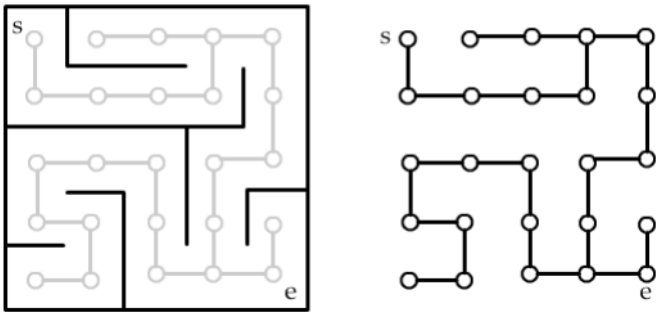
Permasalahan *Maze* merupakan permasalahan yang cukup tua yaitu sekitar 33 tahun yang lalu [1]. Dalam kehidupan sehari – hari *maze* sering dikenal sebagai labirin. Maze atau labirin merupakan suatu saluran atau jaringan yang rumit karena banyak jalan buntu dan persimpangan.



Gambar 1. Perbedaan *perfect* dan *imperfect maze*. [7]

Secara umum terdapat dua jenis *maze*, yaitu *perfect* dan *imperfect maze. Maze* yang tidak terdapat *loop* di dalamnya disebut *perfect maze.* Sebaliknya, *Maze* yang terdapat *loop* di dalamnya disebut *imperfect maze* (lihat Gambar 1).

Terdapat banyak algoitma untuk menyelesaikan permasalahan *maze*, diantaranya algoritma – algoritma tersebut terdapat kesamaan algoritma yang dapat menyelesaikan permasalahan *graph*. Hal ini dikarenakan permasalahan *maze* hampir sama dengan permasalahan *graph.* Sehingga *maze* merupakan permasalahan yang ekuivalen dengan permasalahan/teori *graph*, sebagaimana Gambar 2 yang menunjukkan bahwa *maze* dapat direpresentasikan sebgai *graph.*



Gambar 2. Representasi *maze* sebagai *graph.*[6]

Pada tulisan ini yang akan dianalisis dan diimplementasikan adalah *Wall Follower Algorithm, Depth-First Search Algorithm dan Breadth-First Search Algorithm.*

II. MAZE SOLVER

A. Wall Follower Algorithm

Wall Follower Algorithm merupakan algoritma yang cukup sederhana dan mudah di implementasikan. Dari posisi awal, pemain cukup mengikuti dinding kiri atau kanan dari *maze*. Jika bertemu suatu persimpangan, pemain akan berbelok ke kanan atau ke kiri sesuai dengan dinding yang dia ikuti.

Untuk pemain yang menggunakan dinding kanan sebagai acuan [3]:

- Dimulai dari *starting point*
- Jika tidak ada dinding di sebelah kanan, belok kanan.
- Jika tidak ada dinding di depan, bergerak maju.
- Jika tidak ada dinding di sebalah kiri, belok kiri
- Selain ketiga kondisi diatas, putar balik.

Untuk pemain yang menggunakan dinding kiri sebagai acuan:

- Dimulai dari *starting point*
- Jika tidak ada dinding di sebelah kiri, belok kiri.
- Jika tidak ada dinding di depan, bergerak maju.
- Jika tidak ada dinding di sebalah kanan, belok kanan
- Selain ketiga kondisi diatas, putar balik.

B. *Breadth-First Search Algorithm*

Breadth-First Search Algorithm atau yang lebih sering disebut BFS merupakan algoritma sederhana untuk pencarian/*searching*. Ide dasarnya adalah menjelajahi seleruh *vertex* atau *node* yang terjangkau dari titik awal secara merata. Dalam permasalahan ini *node* atau *vertex* direpresentasikan dalam setiap koordinat pada *maze*. Sehingga untuk setiap *vertex* terdapat empat tetangga yang bisa dikunjungi selanjutnya, yaitu atas, kanan, bawah dan kiri. Berikut adalah alur dari BFS pada permasalahan *maze.*

1. Dimulai dari titik awal
2. *Traverse* ke semua *vertex*/tetangga yang mungkin dikunjungi
3. Dari setiap tetangga cek apakah sudah mencapai tujuan. Jika sudah maka solusi ditemukan.
4. Untuk setiap tetangga, ulangi langkah dua dan tiga.
5. Solusi pertama yang ditemukan merupakan solusi terpendek.

C. *Depth-First Search Algorithm*

Depth-First Search Algorithm atau sering disebut DFS melibatkan *backtracking* untuk pencarian suatu *vertex*. DFS melakukan traversal ke kedalaman terlebih dahulu, sehingga algoritma ini disebut *Depth-First Search. Maze* dapat diselesaikan menggunakan DFS karena *maze* itu sendiri murupakan *graph* (lihat Gambar 2.). Berikut adalah alur penyelesaian *maze* menggunakan DFS:

1. Dimulai dari titik awal
2. Dari semua *vertex* yang dapat dikunjungi, pilih satu untuk dikunjungi. *Vertex* yang dapat dikunjungi merupakan *vertex* yang belum dikunjungi sebelumnya dan bukan dinding penghalang.
3. Tandai *vertex* yang sudah dikunjungi.
4. Jika bertemu jalan buntu, artinya jalur tersebut bukan solusi yang diinginkan. Lakukan *backtracking* dengan kembali pada step sebelumnya hingga terdapat *vertex* yang dapat dikunjungi.
5. Jika bertemu dengan *vertex* tujuan, maka path tersebut merupakan *path* solusi. Lakukan *backtracking* untuk menandai *path* solusi tersebut.

III. IMPLEMENTASI

Dari ketiga *maze solver* yang sudah dibahas, akan diemplementasikan menggunakan bahasa pemrograman Java. Berikut merupakan hasil implementasi *maze solver:*

A. *Wall-Follower Algorithm*

Wall-Follower Algorithm cukup mudah diimplementasikan, ide dasarnya adalah dengan melihat arah pergerakan *pointer* terakhir kali melangkah. Hal ini dilakukan agar program bisa mengenali arah laju *pointer* saat itu sehingga bisa mengenali arah kanan, kiri, maju dan mundur. Lalu simpan pergerakan *pointer* tersebut untuk digunakan pada pergerakan selanjutnya. Untuk lebih jelas lagi, perhatikan ilustrasi yang ditunjukan pada Gambar 3.

Using Right-Hand Rule:				
Facing?	**1st Priority**	**2nd Priority**	**3rd Priority**	**4th Priority**
North	E	N	W	S
South	W	S	E	N
East	S	E	N	W
West	N	W	S	E

Using Left-Hand Rule:				
Facing?	**1st Priority**	**2nd Priority**	**3rd Priority**	**4th Priority**
North	W	N	E	S
South	E	S	W	N
East	N	E	S	W
West	S	W	N	E

Gambar 3. Ilustrasi arah pergerakan *pointer*. [8]

Dari penjelasan tersebut, algoritma *Wall-Follower* dapat diimplementasikan menggunakan rekursif atau iterasi. Namun implementasi menggunakan rekursif memungkinkan terjadi *stack overflow*, yaitu ketika dilakukan pemanggilan *recursive case* secara terus menerus hingga ribuan kali segingga stack yang digunakan menyimpan rekursif bertumpuk banyak dan dianggap *stack overflow* oleh Java. Hal ini terjadi karena *Wall-Follower* akan terus memanggil *recursive case* sampai bertemu dengan titik akhir *maze*. Sehingga algoritma *Wall-Follower* lebih cocok diimplementasikan menggunakan iterasi.

B. *Breadth-First Search Algorithm*

Algoritma BFS dapat di implementasikan dengan *queue.* Dimulai dari titik awal *maze*, masukan vertex titik awal pada *queue*. Lalu mulai iterasi dengan alur sebagai berikut:

1. *Pop* sebuah *vertex* dari *queue.*
2. Cek *vertex* tersebut apakah *vertex* tujuan atau tidak. Jika *vertex* tersebut merupakan *vertex* tujuan maka program selesai, path sudah ditemukan.

3. Masukan semua tetangga yang mungkin dikunjungi dari *vertex* tersebut pada *queue*.
4. Iterasi selesai apabila *queue* kosong.

Perhatikan bahwa alur tersebut hanya dapat menemukan *vertex* tujuan saja namun belum bisa mencari *path* ke *vertex* tujuan tersebut. Untuk mencari *path*, perlu disimpan *previous vertex* pada setiap *vertex*. Untuk menyimpannya, dapat dilakukan pada step 4, yaitu simpan *vertex* sebagai *previous vertex* pada semua tetangga yang dapat dikunjungi pada *vertex* tersebut.

Di akhir program, ketika sudah ditemukan solusi/*vertex* tujuan, maka dapat dicari *path*-nya dari *vertex* tujuan tersebut dengan melakukan *traversal* ke setiap *previous vertex*.

C. *Depth-First Search Algorithm*

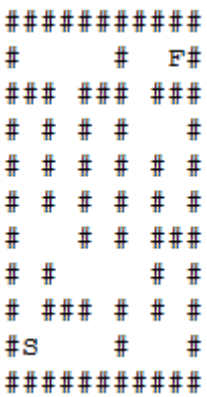
Algoritma DFS mudah diimplementasikan menggunakan rekursif *backtracking*. Implentasinya hampir sama dengan dengan yang sudah dijelaskan pada Bab II, hanya saja perlu ditambahkan penanda ketika sudah bertemu dengan *vertex* tujuan.

IV. EKSPERIMEN

A. *Pengujian*

Untuk menguji algoritma *Wall-Follower*, BFS dan DFS yang telah diimplementasikan, digunakan beberapa *test case*. Pertama – tama, dibuat beberapa *test case* dengan program maze generator dengan algoritma *Prim* [5]. Format test case yang dibuat adalah sebagai berikut:

- Dinding direpresentasikan dengan karakter '#'.
- Titik awal direpresentasikan dengan karakter 'S'.
- Titik akhir direpresentasikan dengan karakter 'F'.
- *Path* solusi direpresentasikan dengan karakter '*'.
- Jalan kosong yang bisa dilalui direpresentasikan dengan karakter ' ' (spasi).

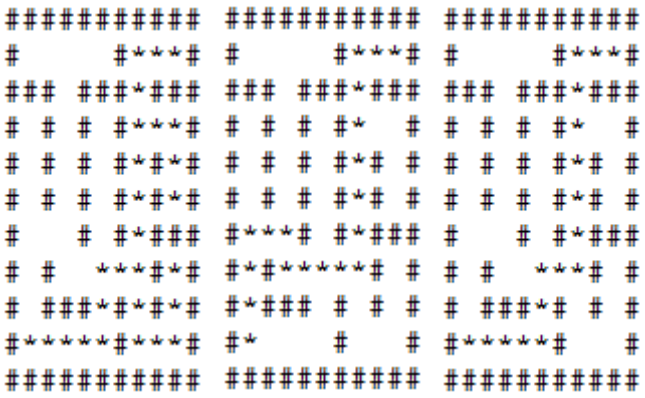


Gambar 4. Contoh *test case* 10x10.

Gambar 4 merupakan contoh input *test case* 10x10. Perhatikan bahwa *maze* tersebut kemungkinan bisa memiliki lebih dari satu solusi.

B. *Output*

Berikut merupakan contoh *output* solusi dari *testcase* 10x10 pada Gambar 4:



Gambar 5. Contoh *output maze* 10x10. Dari kiri ke kanan berturut – turut merupakan solusi dengan algoritma *Wall-Follower*, DFS dan BFS.

C. *Hasil*

Dari hasil pengujian, diperoleh dua tabel perbandingan sebagai berikut:

Tabel 1.
Tabel perbandingan *running time* dalam satuan *millisecond*.

Ukuran *maze*	*Wall-Follower*	DFS	BFS
10x10	0	0	1
20x20	0	0	1
40x40	0	1	2
100x100	2	3	4
200x200	6	5	10
400x400	14	11	28
1000x1000	51	73	96
2000x2000	107	168	283

Tabel 2.
Tabel perbandingan panjang *path*.

Ukuran *maze*	*Wall-Follower*	DFS	BFS
10x10	37	19	17
20x20	317	57	57
40x40	333	77	77
100x100	6985	449	449
200x200	14689	405	405
400x400	41617	969	969
1000x1000	957857	5505	5505
2000x2000	3536397	9793	9793

V. ANALISIS

A. *Wall-Follower Algorithm*

Tabel 1 menunjukkan bahwa *Wall-Follower* merupakan algoritma yang paling *superior* dilihat dari segi *running time*. Hal ini dikarenakan tidak banyak titik/*vertex* yang dikunjungi hingga program menemukan titik tujuan.

Namun, *Wall-Follower* hanya bertujuan untuk membawa *pointer* sampai kepada tujuan saja, tidak mencari *path* sampai pada tujuan. Perhatikan bahwa *path* pada algoritma ini dapat dicari dan dihitung panjangnya. *Path* yang dimaksud adalah sequence titik – titik yang sudah dilalui *pointer* dan panjang *path* didefinisikan sebagai jarak (perhatikan bahwa jarak berbeda dengan perpindahan) dari titik awal hingga titik tujuan yang dilalui oleh *pointer*. Seperti yang ditunjukan oleh Tabel 2, dimana panjang *path* dari algoritma *Wall-Follower* jauh lebih besar diandingkan dengan dua algoritma yang lain.

Perhatikan pula pada tahap implementasi, algoritma *Wall-Follower* di implementasikan dengan iterasi tanpa digunakan struktur data yang membutuhkan memori penyimpanan seperti *queue* dan *stack.* Sehingga bisa dikatakan bahwa algoritma *Wall-Follower* bebas memori, artinya tidak perlu digunakan memori ekstra dalam mengimplementasikannya.

Gambar 6. Ilustrasi test case yang menghasilkan *infinite loop* jika diselesaikan menggunakan algoritma *Wall-Follower*.

Jika dilakukan pengujian lebih lanjut, algoritma *Wall-Follower* tidak selalu memberikan suatu solusi, yaitu pada kasus apabila terdapat sirkuit yang berputar di tengah *maze* dan *pointer start* dimulai pada sirkuit tersebut seperti yang di ilustrasikan pada Gambar 6. Hal ini merupakan salah satu kelemahan dari algoritma *Wall-Follower*, yaitu belum tentu memberikan suatu solusi padahal seharusnya terdapat paling tidak sebuah solusi.

B. *Depth-First Search Algorithm*

Tabel 1 menunjukkan bahwa *running time* algoritma DFS hampir sama dengan *Wall-Follower*, walaupun secara keseluruhan lebih lambat. Terdapat 2 *test case* dimana algoritma DFS lebih cepat dari *Wall-Follower*. Hal ini terjadi karena DFS bergantung pada prioritas pemilihan tetangga yang akan dikunjungi selanjutnya oleh *pointer*. Pada eksperimen kali ini, prioritas yang digunakan secara berurutan adalah atas, bawah, kanan dan kiri.

Gambar 7. Ilustrasi DFS (panah berwarna merah) yang tenggelam terlalu dalam, sedangkan jika menggunakan algoritma *Wall-Follower* (panah berwarna biru) solusi yang ditemukan sangat dekat.

Gambar 8. Ilustrasi penelusuran merata algoritma BFS. Panah biru menunjukkan penelusuran *Wall-Follower* yang tidak merata namun dapat menemukan solusi dengan cepat.

Selain itu, *running time* DFS juga bergantung pada *testcase*, yaitu untuk *testcase* tertentu, DFS dapat tenggelam terlalu dalam. Artinya *pointer maze* mengunjungi *vertex* – *vertex* hingga unjung yang terlalu jauh dengan solusi, padahal *vertex* tujuan lebih dekat jika menggunakan pilihan jalan yang lain seperti yang ditunjukkan oleh Gambar 7.

Berbeda dengan *Wall-Follower*, tujuan DFS merupakan mencari *path* hingga mencapai tujuan *maze*.Hal ini dapat dilihat pada Tabel 2, dimana panjang *path* DFS jauh lebih pendek daripada *Wall-Follower*. Sehingga dapat dikatakan bahwa DFS memperbaiki kelemahan *Wall-Follower* yang hanya berfokus mencari *vertex* tujuan saja namun tidak memperhatikan *path* yang dilaluinya.

Perhatikan bahwa pada implementasi DFS, digunakan *recursive backtracking*, artinya akan digunakan *stack* untuk menyimpan *state recursive* pada program DFS. Oleh karena penggunaan *stack* tersebut, DFS membutuhkan memori penyimpanan, tidak seperti *Wall-Follower* yang tidak membutuhkan memori tambahan.

C. Breadth-First Search Algorithm

Gambar 5 dan Tabel 2 (*testcase* 10x10) menunjukkan bahwa BFS menghasilkan *path* yang paling pendek jika *testcase maze* memiliki dua solusi atau lebih. Hal ini dikarenakan BFS mengunjungi *vertex* selanjutnya dari yang paling dekat dengan titik awal BFS dimulai secara merata. Sehingga *path* yang bertemu dengan *vertex* tujuan pertama kali dijamin merupakan *path* paling pendek. Hal ini sekaligus menunjukkan kelemahan DFS yang belum tentu menghasilkan *path* terpendek, yang kemudian disempurnakan oleh algoritma BFS.

Tabel 1 menujukan *running time* BFS paling lambat dibanding dua algoritma lainnya. Hal ini merupakan *trade off* dari *shortest path* yang diperoleh dari algortima BFS. Untuk memperoleh *shortest path*, BFS harus melakukan penelusuran secara merata dari *vertex* paling dekat dari *vertex* awal dimulainya penelusuran. Gambar 8 menunjukkan bahwa terdapat penelusuran yang sebenarnya tidak perlu dilakukan untuk sampai pada *vertex* tujuan F, sedangkan jika digunakan *Wall-Follower* akan langsung bertemu dengan solusi yang ditunjukan dengan panah biru.

Perhatikan bahwa pada implementasi BFS, digunakan *queue* untuk menyimpan *vertex* yang harus dikunjungi selanjutnya. Sehingga BFS memerlukan memori tambahan untuk penyimpanan *queue* tersebut. Dengan kata lain, BFS tidak *memory free* seperti algoritma *Wall-Follower*.

D. Perbandingan

Dari ketiga analisis algoritma diatas dapat diambil suatu perbandingan dengan jelas yang dapat ditunjukan dengan tabel berikut ini:

Tabel 3. Tabel perbandingan algoritma Wall-Follower, DFS dan BFS pada *maze solver*. [9]

Algo	Solution	Shortest	Guarantee	Fast	Memory
WF	1	No.	No.	Yes	Yes
DFS	1	No.	Yes	Yes	No.
BFS	All	Yes	Yes	Yes	No.

Keterangan:

- *Solution*: banyaknya solusi yang ditemukan.
- *Shortest*: apakah solusi yang ditemukan terjamin merupakan solusi terpendek.
- *Guarantee* : apakah pasti ditemukan solusi atau tidak.
- *Fast:* termasuk cepat atau tidak program yang dijalankan
- *Memory:* apakah algoritma bebas memori tambahan atau tidak.

VI. KESIMPULAN DAN SARAN

Dari pembahasan yang diperoleh pada BAB V, dapat diambil kesimpulan bahwa setiap algoritma yaitu Wall-Follower, BFS dan DFS memiliki kelebihan dan kekurangannya masing – masing. Untuk algoritma *Wall-Follower* kelebihannya adalah *running time* cepat dan *free memory*, sedangkan kekurangannya adalah tidak bisa menemukan *path* yang pendek dan belum tentu menemukan solusi untuk kasus *loop* ditengah *maze*. Untuk algoritma DFS, kelebihannya adalah pasti menemukan solusi baik untuk *perfect* maupun *imperfect maze*. Sedangkan kekurangannya adalah *path* yang dihasilkan belum tentu *path* yang terpendek dan dibutuhkan memori tambahan untuk melakukan *backtracking*. Untuk algoritma BFS, kelebihannya adalah pasti menemukan solusi dengan *path* terpendek. Sedangkan kekurangannya adalah *running time* lebih lama daripada kedua algoritma lain dan tidak *memory free*.

Lalu, penulis menyarankan untuk memperhatikan performa dan hasil yang diinginkan dari algoritma jika ingin memilih algoritma yang tepat dari ketiga algoritma yang sudah dibahas. Misalnya, ingin ditemukan *shortest path*, maka algoritma yang cocok adalah BFS.

VII. REFERENSI

[1] M. Alsubaie, “Algorithms for Maze Solving Robot,” Manchester, 2013.

[2] D. B. Suits, “Playing With Mazes,” Department of Philosophy Rochester Institute of Technology , Rochester NY, 1994.

[3] J. Stilley, “Github - The mazelib API,” 2013. [Online]. Available: https://github.com/theJollySin/mazelib. [Diakses 18 November 2016].

[4] T. H. Cormen, C. E. Leiserson, R. L. Rivest dan C. Stein, Introduction to Algorithms, Cambridge, Massachusetts;London, England: The MIT Press, 2009.

[5] S. Silverman, “The Source for Java Technology Collaboration,” Oracle Corporation, 1 Maret 2008. [Online]. Available: https://java.net/projects/trackbot-greenfoot/sources/svn/show/trunk/Version1/TrackBot/maze. [Diakses 25 November 2016].

[6] D. K. Wind, “Embedding Mazes - Spanning Trees of a Grid Graph,” Stack Exchange Inc, [Online]. Available: http://math.stackexchange.com/questions/119344/embedding-mazes-spanning-trees-of-a-grid-graph. [Diakses 27 November 2016].

[7] M. j, “The Perfect Maze,” flylib.com, 2008. [Online]. Available: http://flylib.com/books/en/2.760.1.66/1/. [Diakses 27 November 2016].

[8] B. Carey, “Maze Solving Algorithms – Wall Follower,” [Online]. Available: http://avian.netne.net/blog/?p=93. [Diakses 28 November 2016].

[9] W. D. Pullen, “Maze Classification,” 14 Juli 1996. [Online]. Available: http://www.astrolog.org/labyrnth/algrithm.htm. [Diakses 28 November 2016].