Pemanfaatan Graf dalam Menentukan Solusi dari Gambar Teka Teki Labirin Kotak

Muhammad Khairul Makirin 13517088
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
khairul240999@gmail.com

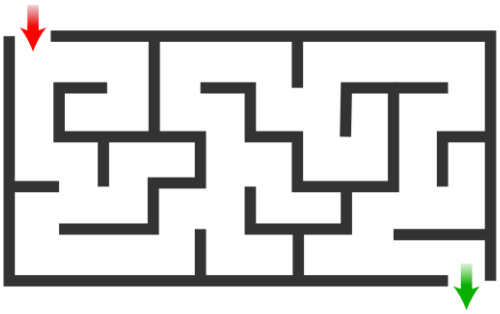
Abstrak — Graf merupakan suatu struktur data yang banyak digunakan di berbagai bidang seperti matematika, komputer, dan biologi. Salah satu pemanfaatan graf pada bidang komputer adalah *mapping* atau memetakan suatu tempat dan navigasi dari tempat ke tempat. Sebuah teka teki labirin dapat dikatakan sebagai suatu representasi sederhana dari peta atau tempat di dunia nyata, karena itu makalah ini akan membahas bagaimana memetakan dan menavigasikan atau mencari solusi sebuah teka teki labirin kotak sederhana dengan bantuan graf.

Kata kunci — Algoritma pencarian pada graf, graf, labirin, manipulasi gambar, membentuk graf dari gambar.

I. PENDAHULUAN

Graf adalah suatu diagram yang terdiri atas simpul dan relasi yang dimiliki antar simpul. Dalam matematika graf didefinisikan sebagai kumpulan / himpunan dari simpul dan sisi, yang mana suatu sisi menghubungkan suatu simpul dengan simpul yang lain, dalam hal ini sisi dapat diibaratkan sebagai suatu relasi yang menghubungkan dua buah simpul[1].

Dalam bidang ilmu komputer, graf sering sekali dipakai sebagai struktur data relasional, yaitu suatu struktur yang menggambarkan relasi antara dua buah tipe data. Dalam pengaplikasiannya graf digunakan dalam banyak pemecahan masalah seperti basis data, navigasi, dan pemetaan suatu tempat, salah satu contoh dari pemetaan adalah memetakan sebuah gambar teka teki labirin menggunakan algoritma khusus dan membentuk representasi graf dari labirin, dengan ini representasi graf tersebut dapat dipakai untuk menavigasikan dan menyelesaikan teka teki labirin tersebut.



Gambar 1. Teka teki labirin
(Sumber: https://en.wikipedia.org/wiki/Maze)

II. DEFINISI

A. Graf

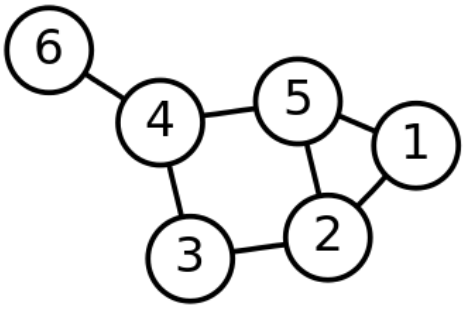
Dalam matematika graf didefinisikan sebagai pasangan himpunan simpul dan sisi[1],

$$G = (V, E)$$
$$V = \{v_1, v_2, \dots, v_n\}$$
$$E = \{e_1, e_2, \dots, e_n\}$$

di mana V adalah himpunan tidak kosong dari simpul-simpul, dan E adalah himpunan sisi yang mengubungkan sebuah dua buah simpul.

Dalam penggunaannya graf dapat dikelompokkan menjadi beberapa jenis yaitu[1] :

1. Graf sederhana
2. Graf tidak sederhana
3. Graf berhingga
4. Graf tidak berhingga
5. Graf berarah
6. Graf tidak berarah
7. Graf kosong, dst.



Gambar 2. Graf sederhana dengan 6 simpul
(Sumber: https://medium.freecodecamp.org/a-gentle-introduction-to-data-structures-how-graphs-work-a223d9ef8837)

Dalam teori graf, ada beberapa terminologi / istilah yang sering ditemui yaitu[1]:

1. **Bertetanggaan (*adjacent*)**
   Dua buah simpul dikatakan bertetanggaan jika

kedua simpul dihubungkan langsung dengan sebuah sisi.

2. **Bersisian (*incident*)**
   Untuk sembarang sisi $e = (v_i, v_j)$, e dikatakan bersisian dengan simpul v_i dan v_j.
3. **Simpul terpencil (*isolated vertex*)**
   Simpul terpencil adalah simpul yang tidak mempunyai sisi yang bersisian dengan simpul tersebut.
4. **Lintasan (*path*)**
   Sebuah lintasan adalah barisan berselang seling antara simpul dan sisi, sebagai contoh dalam gambar 2 terdapat lintasan dari simpul 6 ke simpul 4 yaitu 6, e_{46}, dan 4.
5. **Derajat (*degree*)**
   Derajat menyatakan banyaknya sisi yang bersisian dengan suatu simpul, contoh pada gambar 2 derajat simpul 6 adalah 1 dan derajat simpul 4 adalah 3.
6. **Siklus (*cycle*) atau Sirkuit (*circuit*)**
   Lintasan yang berawal dan berakhir di simpul yang sama disebut suatu sirkuit
7. **Terhubung (*connected*)**
   Dua buah simpul dikatakan terhubung jika ada lintasan antara dua simpul tersebut
8. **Upagraf (*subgraf*)**
   Jika G adalah sebuah graf, $G = (V, E)$, maka $G_1 = (V_1, E_1)$ merupakan sebuah upagraf (*subgraf*) dari G jika $V_1 \subseteq V$ dan $E_1 \subseteq E$.

B. Teka teki Labirin

Sebuah teka teki labirin merupakan sebuah teka teki yang mempunyai titik awal (*start*), titik akhir (*finish*) dan jalur-jalur yang dapat ditelusuri dari titik awal[3]. Inti dari teka teki tersebut adalah untuk mencari jalur yang menghubungkan titik awal dan titik akhir.

III. ALGORITMA PENCARIAAN PADA GRAF

Ada banyak variasi algoritma pencarian pada graf, beberapa contoh yang akan dibahas adalah *Breadth First Search* dan *Depth First Search*.

A. Breadth First Search

Breadth First Search adalah algoritma pencarian pada graf yang memanfaatkan struktur data *Queue* untuk memproses tiap simpul dan sebuah daftar simpul-simpul yang sudah dikunjungi agar tidak mengunjungi simpul yang sama dua kali[4]. Kompleksitas waktu dari algoritma ini adalah $O(V + E)$[5], dimana V adalah banyaknya simpul dan E adalah banyaknya sisi. Dengan asumsi tiap simpul terhubung ke satu atau lebih simpul lain langkah-langkahnya adalah sebagai berikut:

1. Tentukan simpul pertama atau simpul mulai, dan tentukan simpul yang ingin dicari
2. Jika simpul sekarang bukan simpul yang ingin dicari, lanjutkan ke langkah 3, jika ya, lanjutkan ke langkah 7
3. Masukkan simpul sekarang ke dalam daftar simpul yang sudah dilewati dan *Queue* (simpul sekarang menjadi *first queue*)
4. Pilih satu simpul yang bertetanggaan dengan simpul sekarang dan belum dikunjungi, jika simpul tersebut bukan simpul yang ingin dicari, masukkan simpul tersebut ke dalam *Queue* dan daftar simpul yang sudah dikunjungi, jika tidak ada simpul yang belum dikunjungi, ubah simpul sekarang menjadi simpul setelah *first queue* dan *first queue* menjadi simpul sekarang, jika simpul tersebut adalah simpul yang ingin dicari, lanjutkan ke langkah 7
5. Ulangi langkah 4 selama tiap simpul yang bertetanggaan dengan simpul sekarang belum dimasukkan ke *Queue*, simpul yang ingin dicari belum ditemukan, dan *Queue* tidak kosong
6. Ubah *first queue* menjadi elemen setelahnya, kemudian ulangi langkah 4 selama belum semua simpul dikunjungi dengan simpul sekarang adalah *first queue* yang baru
7. Akhir dari proses

```
BFS(G, s)
 1  for each vertex u   V [G] - {s}
 2       do color[u] ← WHITE
 3          d[u] ← ∞
 4          π[u] ← NIL
 5  color[s] ← GRAY
 6  d[s] ← 0
 7  π[s] ← NIL
 8  Q ← Ø
 9  ENQUEUE(Q, s)
10  while Q ≠ Ø
11       do u ← DEQUEUE(Q)
12          for each v   Adj[u]
13              do if color[v] = WHITE
14                    then color[v] ← GRAY
15                         d[v] ← d[u] + 1
16                         π[v] ← u
17                         ENQUEUE(Q, v)
18          color[u] ← BLACK
```

Gambar 3. Pseudocode untuk Breadth First Search
(Sumber: http://web.ist.utl.pt/fabio.ferreira/material/asa/clrs.pdf)

B. Depth First Search

Sedikit berbeda dengan algoritma pencarian *Breadth First Search*, algoritma *Depth First Search* menggunakan struktur data *Stack* dalam memproses tiap simpul[4]. Kompleksitas waktu dari algortima ini sama seperti *Breadth First Search* yaitu $O(V + E)$[5]. Dengan asumsi yang sama seperti di *Breadth First Search* langkah-langkahnya adalah sebagai berikut:

1. Tentukan simpul pertama atau simpul mulai, dan tentukan simpul yang ingin dicari
2. Jika simpul sekarang bukan simpul yang ingin dicari, lanjutkan ke langkah 3, jika ya, lanjutkan ke langkah 5
3. Masukkan simpul sekarang ke dalam *Stack* (simpul sekarang menjadi *top stack*) dan daftar simpul yang telah dikunjungi

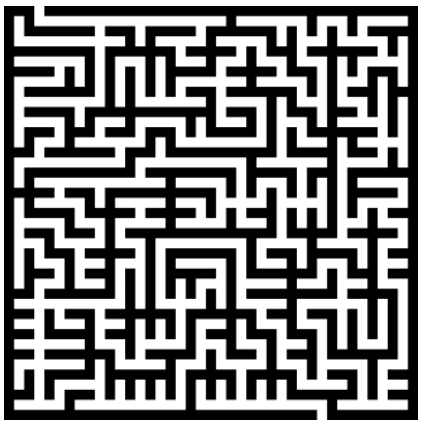
4. Pilih satu simpul yang bertetanggaan dengan simpul sekarang dan belum dikunjungi, jika tidak ada simpul yang belum dikunjungi *pop stack* dan ubah simpul sekarang menjadi elemen *top stack*, jika ada simpul yang belum dikunjungi, ulangi langkah 2 dengan simpul tersebut sebagai simpul sekarang selama *Stack* tidak kosong
5. Akhir dari proses

```
DFS(G)
1  for each vertex u   V [G]
2       do color[u] ← WHITE
3          π[u] ← NIL
4  time ← 0
5  for each vertex u   V [G]
6       do if color[u] = WHITE
7             then DFS-VISIT(u)
DFS-VISIT(u)
1  color[u] ← GRAY     ▷White vertex u has just been discovered
2  time ← time +1
3  d[u] time
4  for each v   Adj[u]  ▷Explore edge(u, v).
5       do if color[v] = WHITE
6             then π[v] ← u
7                  DFS-VISIT(v)
8  color[u] BLACK      ▷ Blacken u; it is finished.
9  f [u] ▷ time ← time +1
```

Gambar 4. Pseudocode untuk Depth First Search
(Sumber:
http://web.ist.utl.pt/fabio.ferreira/material/asa/clrs.pdf)

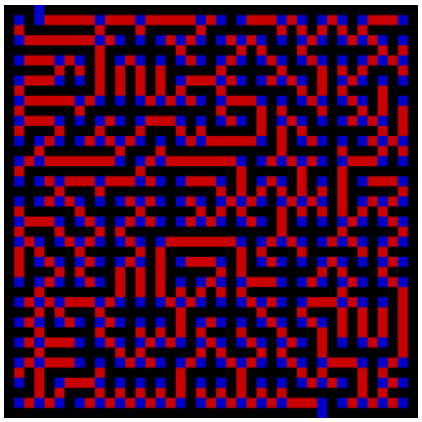
IV. Cara Kerja

Dalam menentukan solusi dari gambar teka teki labirin dengan bantuan graf, gambar tersebut harus diubah terlebih dahulu menjadi representasi grafnya. Tipe graf yang akan dipakai adalah graf sederhana, sebuah graf dikatakan graf sederhana jika graf tersebut tidak mengandung gelang dan sisi-ganda[1], sebuah graf disebut memiliki gelang jika terdapat sebuah sisi yang menghubungkan satu simpul yang sama, sedangkan graf yang mengandung sisi-ganda adalah graf yang memuat lebih dari dua buah sisi antara simpul v_i dan v_j. Pada algoritma penentuan solusi dari gambar labirin yang penulis buat, penulis merepresentasikan graf dalam bentuk senarai ketetanggaan (*Adjacency List*)[1], di mana setiap simpulnya menyimpan data berupa posisi simpul tersebut pada gambar dan apakah di kanan, kiri, atas, atau bawah simpul terdapat jalur, di bawah ini adalah contoh gambar labirin yang diubah menjadi representasi grafnya.



Gambar 5. Contoh labirin

(Sumber: https://github.com/mikepound/mazesolving)



Gambar 6. Representasi graf dari gambar 5

pada gambar 5, jalur dari labirin adalah garis yang berwarna putih, sedangkan dinding labirin adalah garis yang berwarna hitam, kemudian dapat dilihat pada gambar 6, kotak warna biru menggambarkan simpul yang dibuat, sedangkan garis warna merah menggambarkan sisi yang dibuat.

Ada dua langkah yang penting untuk mencari solusi dari teka teki labirin dengan graf yaitu membentuk representasi graf dari labirin dan menyelesaikan labirin tersebut dengan menavigasi graf yang telah dibentuk[2].

A. Membentuk Representasi Graf dari Labirin

Langkah pertama dalam membentuk representasi graf dari gambar labirin adalah menentukan posisi simpul-simpul yang tepat sesuai dengan bentuk labirinnya. Hal ini dapat dicapai dengan cara menaruh simpul pada tiap persimpangan atau belokan, untuk menentukan apakah pada posisi tersebut terdapat persimpangan atau belokan, dapat dilihat dari keempat sisi di posisi tersebut yaitu atas, kanan, kiri, dan bawah, jika terdapat jalur di atas dan bawah saja atau di kanan dan kiri saja, maka di posisi tersebut tidak terdapat persimpangan atau belokan, sebaliknya maka di posisi tersebut terdapat persimpangan atau belokan. Menentukan simpul pada sebuah gambar labirin dilakukan secara mengiterasi tiap piksel pada gambar dari kiri atas ke kanan bawah, kemudian simpul dimasukkan ke dalam senarai ketetanggaan.

Langkah kedua adalah membentuk sisi dari simpul yang sudah dibuat. Sebuah sisi dibuat dari dua buah simpul, karena dalam menentukan sebuah simpul memakai iterasi tiap piksel dari kiri atas ke kanan bawah gambar, untuk tiap simpul baru yang diletakkan hanya perlu melihat apakah di atas dan kiri simpul baru ini terdapat simpul lain yang lintasan antara kedua simpul ini tidak dihalangi oleh dinding labirin, kemudian membuat sisi berdasarkan kondisi di atas dan memperbarui senarai ketetanggaan.

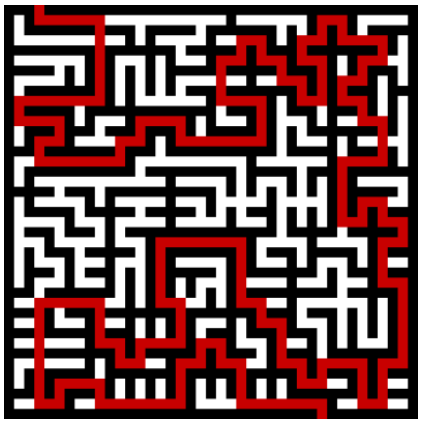
Langkah ketiga adalah menentukan titik awal dan akhir. Dalam gambar 3, titik awal labirin berada di atas sedangkan titik akhir labirin berada di bawah, secara umum jika labirin mempunyai hanya satu titik awal dan titik akhir, titik awal labirin adalah simpul pertama di senarai, sedangkan titik akhir labirin adalah simpul terakhir yang di senarai.

B. Menyelesaikan Teka teki Labirin

Setelah menentukan posisi simpul dan sisi yang sesuai dengan bentuk labirin, menentukan titik awal dan akhir, dan memasukkannya ke dalam senarai ketetanggaan, langkah selanjutnya adalah menyelesaikannya dengan algoritma pencarian pada graf.

Algoritma untuk mencari sebuah jalur yang menghubungkan titik awal dan akhir sama seperti algoritma pencarian pada graf tetapi dengan menambahkan suatu daftar untuk menyimpan lintasan yang dilewati dari titik akhir. Algoritma pencarian yang mudah untuk diimplementasikan dalam graf dengan senarai ketetanggaan adalah *Breadth First Search* atau *Depth First Search*.

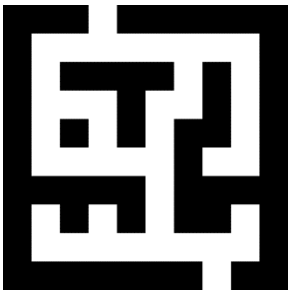
Pada contoh dibawah penulis memakai *Depth First Search* dengan modifikasi tertentu yaitu untuk setiap pemilihan simpul yang bertetanggaan, jarak dari simpul tersebut ke titik akhir juga diperhitungkan, sehingga pemilihan simpul yang bertetanggaan didasarkan pada seberapa dekat simpul tersebut terhadap titik akhir dan simpul yang terdekat akan dipilih untuk dimasukkan ke dalam *Stack*.



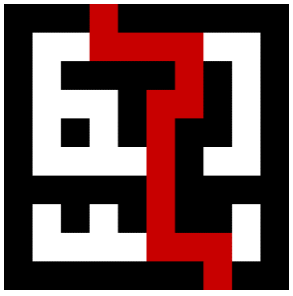
Gambar 7. Hasil pencarian jalur dari gambar 5

V. HASIL PERCOBAAN

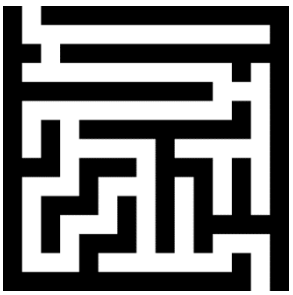
Hasil yang diperoleh dari percobaan berbagai graf mulai dari kecil ke besar adalah sebagai berikut:



Gambar 8. Labirin 1
(Sumber: https://github.com/mikepound/mazesolving)

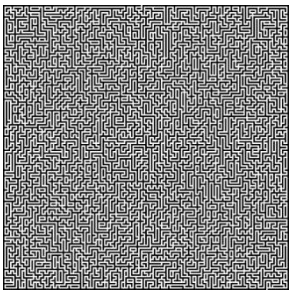


Gambar 9. Hasil pencarian jalur dari gambar 8

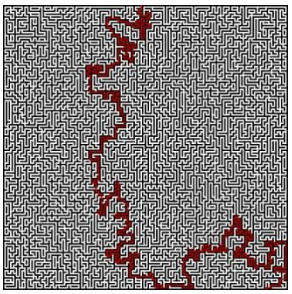


Gambar 10. Labirin 2 (Sumber: https://github.com/mikepound/mazesolving)

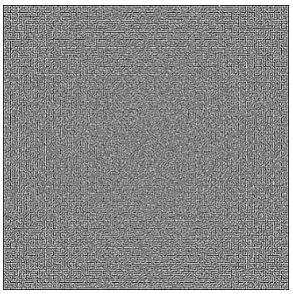
Gambar 11. Hasil pencarian jalur dari gambar 10



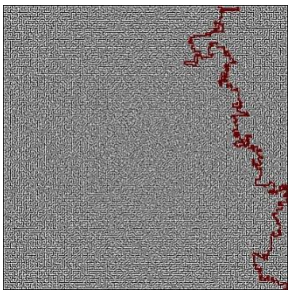
Gambar 12. Labirin 2
(Sumber: https://github.com/mikepound/mazesolving)



Gambar 13. Hasil pencarian jalur dari gambar 12



Gambar 14. Labirin 3
(Sumber: https://github.com/mikepound/mazesolving)



Gambar 15. Hasil pencarian jalur dari gambar 14

VI. KODE SUMBER

Dibawah ini merupakan kode sumber untuk mencari solusi dari gambar teka teki labirin dalam bahasa python.

```
import cv2
import numpy as np
import time
import math

def drawPath(dataNodes, solveStack,
mazeImg) :
    ## Menggambarkan lintasan yang sudah
selesai
    lenStack = len(solveStack)
    i = 0

    while(i + 1 < lenStack) :
        currPos =
(dataNodes[solveStack[i]][0],
dataNodes[solveStack[i]][1])
        nextPos = (dataNodes[solveStack[i
+ 1]][0], dataNodes[solveStack[i + 1]][1])
        cv2.line(mazeImg, currPos,
nextPos, (0, 0, 200), 1)
        i += 1

def drawGraph(dictNodes, dataNodes,
mazeImg) :
   ## Inisialisasi Node source pertama
   currSrcNode = 0
   lenDict = len(dictNodes)

   while(currSrcNode < lenDict) :
       ## Posisi Node source
       currSrcPos =
(dataNodes[currSrcNode][0],
dataNodes[currSrcNode][1])
       i = 0
       while(i <
len(dictNodes[currSrcNode])) :
           ## Node tujuan yang terhubung
dengan node source
           currDstNode =
dictNodes[currSrcNode][i]
           currDstPos =
(dataNodes[currDstNode][0],
dataNodes[currDstNode][1])

           ## Warnai edges
           cv2.line(mazeImg, currSrcPos,
currDstPos, (0, 0, 200), 1) # Merah

           ##Warnai nodes
           cv2.line(mazeImg, currSrcPos,
currSrcPos, (200, 0, 0), 1) # Biru
           cv2.line(mazeImg, currDstPos,
currDstPos, (200, 0, 0), 1) # Biru
           i += 1
       currSrcNode += 1

def searchNode(dictNodes, identity) :
    ## Mencari node dalam dictionary node
    isFound = False

    if identity in dictNodes :
        isFound = True

    return isFound

def addNode(dictNodes, dataNodes,
identity, posX, posY, way1, way2, way3,
way4, visited) :
    ## Menambahkan node ke list adjency
dan list data node-node
    dataNodes[identity] = [posX, posY,
way1, way2, way3, way4, visited]
    dictNodes[identity] = []

def searchEdges(dictNodes, identity1,
identity2) :
    ## Mencari apakah sudah ada sisi
antara node 1 dan node 2
    isFound = False

    if(searchNode(dictNodes, identity1)
and searchNode(dictNodes, identity2)) :
        for identity in dictNodes:
            if(identity == identity1) :
                for adjIdentity in
dictNodes[identity] :
                    if(adjIdentity ==
identity2) :
                        isFound = True
                        break
                if(isFound) :
                    break
            elif(identity == identity2) :
                for adjIdentity in
dictNodes[identity] :
                    if(adjIdentity ==
identity1) :
                        isFound = True
                        break
                if(isFound) :
                    break

    return isFound

def addEdge(dictNodes, identity1,
identity2) :
    ## Menambahkan Hubungan antara dua
node yang sudah ada di dict nodes
    dictNodes[identity1].append(identity2)
    dictNodes[identity2].append(identity1)

def isJunction(mazeImg, currX, currY) :
    ## Menentutkan apakah suatu
kotak/pixel dalam gambar tersebut
merupakan tempat yang pas untuk menaruh
sebuah node baru
    lenRow = len(mazeImg)
    lenCol = len(mazeImg[lenRow-1])
```

```
    if(np.any(mazeImg[currX][currY]) !=
0):
        wayDown = True if(currX + 1 <
lenRow and np.any(mazeImg[currX+1][currY]
!= 0)) else False
        wayUp = True if(currX != 0 and
np.any(mazeImg[currX-1][currY] != 0)) else
False
        wayRight = True if(currY + 1 <
lenCol and np.any(mazeImg[currX][currY+1]
!= 0)) else False
        wayLeft = True if(currY != 0 and
np.any(mazeImg[currX][currY-1] != 0)) else
False

        if(not ((wayLeft and wayRight and
not (wayDown or wayUp)) or (wayUp and
wayDown and not (wayLeft or wayRight)))) :
            return True
        else :
            return False
    else :
        return False

def upperNodeOf(dataNodes, nodeId) :
    ## Mencari node atas terdekat (satu
kolom) dari nodeId
    isFound = False
    upNodeId = None
    enum = nodeId

    thisNode = dataNodes[nodeId]
    thisNodeY = thisNode[0]
    enum -= 1

    while(not isFound and enum >= 0):
        if(dataNodes[enum][0] ==
thisNodeY):
            upNodeId = enum
            isFound = True
        else :
            enum -= 1

    return upNodeId

def leftNodeOf(dataNodes, nodeId) :
    ## Mencari node kiri (satu baris)
terdekat dari nodeId
    isFound = False
    leftNodeId = None
    enum = nodeId

    thisNode = dataNodes[nodeId]
    thisNodeX = thisNode[1]
    enum -= 1

    while(not isFound and enum >= 0):
        if(dataNodes[enum][1] ==
thisNodeX):
            leftNodeId = enum
            isFound = True
        else :
            enum -= 1

    return leftNodeId

def nearEndNode(dataNodes, adjList,
endNode) :
    ## Mencari node dari dari adjList yang
paling dekat dengan titik akhir
    distance =
math.sqrt((dataNodes[endNode][0]-
dataNodes[adjList[0]][0])**2 +
(dataNodes[endNode][1]-
dataNodes[adjList[0]][1])**2)
    node = adjList[0]
    i = 1
    lenAdj = len(adjList)
    while i < lenAdj :
        currDistance =
math.sqrt((dataNodes[endNode][0]-
dataNodes[adjList[i]][0])**2 +
(dataNodes[endNode][1]-
dataNodes[adjList[i]][1])**2)
        if distance > currDistance :
            distance = currDistance
            node = adjList[i]

        i += 1

    return node

def solve():
    ### KAMUS
    dataNodes = {}          # Isinya data
posisiX, posisiY, up, down, left, right
    dictNodes = {}          # Isinya
adjency list
    solveStack = []         # Dipakai
untuk depth first search
    identity = 0            # Inisialisasi
ID Node pertama


    ### ALGORITMA
    ## Mempersiapkan image untuk di-read
    picName = input("File name : ")
    mazeImg = cv2.imread(picName,
cv2.IMREAD_COLOR)

    ## Mempersiapkan timer dan variable
lain
    startTimer = time.process_time()
    lenRow = len(mazeImg)
    lenCol = len(mazeImg[lenRow-1])

    ## Membuat graf dari gambar labirin
    for i in range(0, lenRow) :
        for j in range(0, lenCol) :
            ## Menentukan apakah pada
posisi i, j terdapat sebuah junction, jika
ya maka buat node di situ
            if(isJunction(mazeImg, i, j))
:
```

```
                ## Menentukan apakah dari node tersebut terdapat jalan ke atas, ke samping, atau ke bawah. Jika terdapat jalan hasilnya 1 (True)
                UP = 1 if(i != 0 and np.any(mazeImg[i-1][j] != 0)) else 0
                DOWN = 1 if(i + 1 < lenRow and np.any(mazeImg[i+1][j] != 0)) else 0
                LEFT = 1 if(j != 0 and np.any(mazeImg[i][j-1] != 0)) else 0
                RIGHT = 1 if(j + 1 < lenCol and np.any(mazeImg[i][j+1] != 0)) else 0

                ## Menambahkan node di posisi tersebut
                addNode(dictNodes, dataNodes, identity, j, i, UP, DOWN, LEFT, RIGHT, False)             # j, i karena saat di gambarkan, j adalah kolom (X) dan i adalah baris (Y)

                ## Membentuk edges dari node yang baru dibuat dengan node di atasnya dan/atau di kirinya
                # Jika ada jalan ke atas
                if(UP == 1) :
                    upperNode = upperNodeOf(dataNodes, identity)
                    if(upperNode != None) :
                        addEdge(dictNodes, identity, upperNode)
                # Jika ada jalan ke kiri
                if(LEFT == 1) :
                    leftNode = leftNodeOf(dataNodes, identity)
                    if(leftNode != None) :
                        addEdge(dictNodes, identity, leftNode)

                identity += 1
    ## Menentukan Start Node dan End Node dari data nodes
    startNode = 0
    endNode = len(dataNodes) - 1


    ## Bagian untuk solvingnya (pakai depth first search (stack) ditambah sedikit elemen dari pencariaan jalur terpendek)
    currNode = startNode
# Current Node
    adjNodes = dictNodes[currNode]
# List of adjacent and unvisited Node from current node
    solveStack.append(currNode)
# Stack to process serve as a path too leading from start to finish
    lenSolve = 1
# Untuk optimisasi pencarian
    dataNodes[currNode][6] = True
# Changging current node to visited

    while(currNode != endNode) :

        while(adjNodes != [] and currNode != endNode) :
            nextNode = nearEndNode(dataNodes, adjNodes, endNode)

            ## Jika nextNode belum pernah dikunjungi
            if(not (dataNodes[nextNode][6])) :

                dictNodes[currNode].remove(nextNode)
                currNode = nextNode
                dataNodes[nextNode][6] = True

                solveStack.append(currNode)
                lenSolve += 1
                adjNodes = dictNodes[currNode]

            ## Jika nextNode sudah pernah dikunjungi
            else :

                dictNodes[currNode].remove(nextNode)
                adjNodes = dictNodes[currNode]

        ## Jika tidak ada simpul yang bertetanggaan dengan currNode atau semua simpul yang bertetanggaan sudah pernah dikunjungi
        if(lenSolve > 0 and currNode != endNode) :
            lastNode = solveStack.pop(lenSolve - 1)
            lenSolve -= 1
            currNode = solveStack[lenSolve - 1]
            adjNodes = dictNodes[currNode]
        else :
            break

    ## Jika ditemukan lintasan dari titik awal ke titik akhir
    if(lenSolve > 0) :

        ## Menghentikan timer
        endTimer = time.process_time()
        print("Elapsed Time : ", end="", flush=True)
        print(endTimer - startTimer)

        ## Menggambar hasil solusi di gambar labirinnya
```

```
        drawPath(dataNodes, solveStack,
mazeImg)

        ## Me-resize gambar agar mudah
dilihat
        dim = (400, 400)
        mazeImg = cv2.resize(mazeImg, dim,
interpolation = cv2.INTER_AREA)
        imgBefore = cv2.imread(picName,
cv2.IMREAD_COLOR)
        imgBefore = cv2.resize(imgBefore,
dim, interpolation = cv2.INTER_AREA)

        ## Menampilkan gambar ke layar
        cv2.imshow("after", mazeImg)
        cv2.imshow("before", imgBefore)
        cv2.waitKey(0)
        cv2.destroyAllWindows()
        cv2.imwrite("rez" + picName,
imgBefore)
        cv2.imwrite("hasilpath" + picName,
mazeImg)

    ## jika tidak ditemukan lintasan dari
titik awal ke titik akhir
    else :
        print("There\'s no way from start
to finish")

solve()
```

VII. Kesimpulan

Graf adalah struktur data yang banyak digunakan di bidang computer, salah satu pemanfaatannya adalah untuk *mapping* dan navigasi sebuah tempat. Dalam makalah ini telah dibahas bagaimana caranya untuk memetakan dan mencari solusi dari suatu teka teki labirin yaitu membentuk representasi graf dari gambar labirin dengan menempatkan simpul pada belokan atau persimpangan pada labirin, kemudian menavigasikan graf yang telah dibuat dengan algoritma pencariaan untuk mendapatkan solusi dari teka teki labirin tersebut.

Seperti yang dapat dilihat pada gambar 15, algoritma ini dapat digunakan untuk labirin kotak yang lebih kompleks dan membutuhkan waktu yang lebih banyak jika dikerjakan oleh tangan.

VIII. Ucapan Terima Kasih

Penulis ingin mengucapkan terima kasih kepada Tuhan YME, orang tua, teman, dan semua pihak yang secara langsung maupun tidak langsung membantu kelancaran penulisan makalah ini.

Daftar Pustaka

[1] Munir, Rinaldi, Matematika Diskrit, Bandung: Informatika Bandung, 2009.

[2] Aqel, Mohammad & Issa, Ahmed & Khdair, Mohammed & ElHabbash, Majde & AbuBaker, Mohammed & Massoud, Mohammed, Intelligent Maze Solving Robot Based on Image Processing and Graph Theory Algorithms. ICPET, 2017, pp 48-53.

[3] https://www.usi.edu/media/4024497/Mazes-vs-Labyrinths2.pdf diakses pada 8 Desember 2018 18.30.

[4] https://www.geeksforgeeks.org/graph-data-structure-and-algorithms/#introDFSnBFS diakses pada 8 Desember 2018 18.00

[5] https://www.quora.com/What-is-the-time-complexity-of-Breadth-First-Search-Traversal-of-a-graph diakses pada 8 Desember 2018 10.00

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

Muhammad Khairul Makirin (13517088)