Laporan Praktikum Kecerdasan Buatan

Semester Ganjil 2024/2025

Jurusan Teknik Elektro
Program Studi S1 Teknik Elektro
Fakultas Teknik
Universitas Tidar

Praktikum ke- : 3
Judul Praktikum : Pencarian Tidak Terarah: BFS dan DFS

Nama : Jerico Christianto
NIM : 2220501082

“Laporan praktikum ini saya kerjakan dan selesaikan dengan sebaik-baiknya tanpa melakukan tindak kecurangan. Apabila di kemudian hari ditemukan adanya kecurangan pada laporan ini, maka saya bersedia menerima konsekuensinya.”

Tertanda,

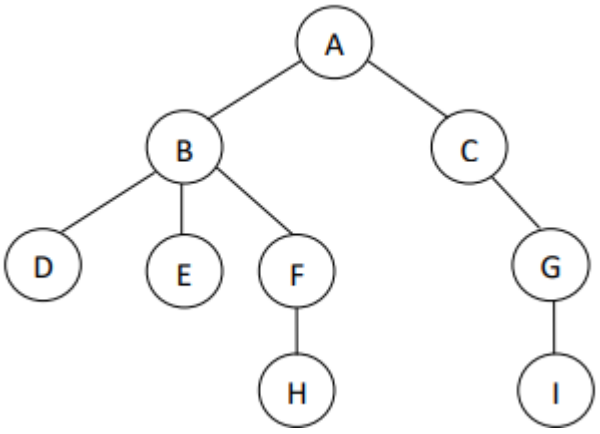
Jerico Christianto

2220501082

A. Tujuan Praktikum
   1. Memahami Konsep Dasar Kecerdasan Buatan
   2. Memahami Konsep Dasar Pencarian Tidak Terarah: BFS dan DFS
   3. Menerapkan Algoritma Pencarian Tidak Terarah: BFS dan DFS

B. Langkah Praktikum
   1. Buka aplikasi JupyterLab

   2. Membuat ilustrasi pencarian



Berdasarkan ilustrasi berikut akan dibuat kode program untuk melakukan pencarian dari satu node ke node yang lain dengan menerapkan algoritma DFS. Untuk menjalankan algoritma DFS dibutuhkan:

- Variabel untuk menampung node yang telah dikunjungi
- Variabel untuk menampung node yang akan dikunjungi

3. Buat kode program untuk pencarian dengan algoritma DFS

```
[1]: graph = {
        'A' : ['B', 'C'],
        'B' : ['D', 'E', 'F'],
        'C' : ['G'],
        'D' : [],
        'E' : [],
        'F' : ['H'],
        'G' : ['I'],
        'H' : [],
        'I' : [],
    }

    def dfs(graph, node):
        visited = []
        stack = []

        visited.append(node)
        stack.append(node)

        while stack:
            s = stack.pop()
            print(s, end = " ")

            for n in reversed(graph[s]):
                if n not in visited:
                    visited.append(n)
                    stack.append(n)

    def main():
        dfs(graph,'A')

    main()
```

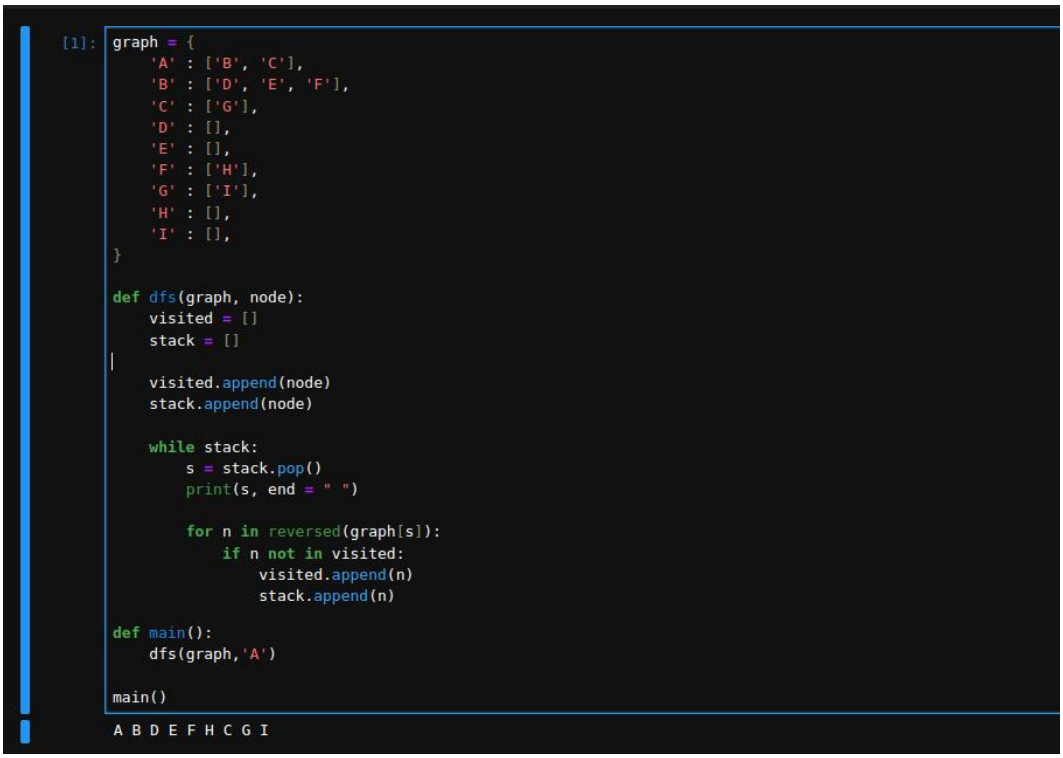
4. Dengan ilustrasi yang sama akan digunakan algoritma BFS untuk pencarian nodenya. Untuk menjalankan algoritma BFS dibutuhkan:
   - Variabel untuk menampung node yang telah dikunjungi
   - Variabel untuk menampung node yang akan dikunjungi
5. Buat kode program untuk pencarian dengan algoritma BFS

```
[2]: from collections import deque

    graph = {
        'A' : ['B', 'C'],
        'B' : ['D', 'E', 'F'],
        'C' : ['G'],
        'D' : [],
        'E' : [],
        'F' : ['H'],
        'G' : ['I'],
        'H' : [],
        'I' : [],
    }

    def bfs(graph, node):
        visited = []
        queue = deque()

        visited.append(node)
        queue.append(node)

        while queue:

            s = queue.popleft()
            print(s, end = " ")

            for n in graph[s]:
                if n not in visited:
                    visited.append(n)
                    queue.append(n)

    def main():
        bfs(graph,'A')

    main()
```

C. Hasil Praktikum

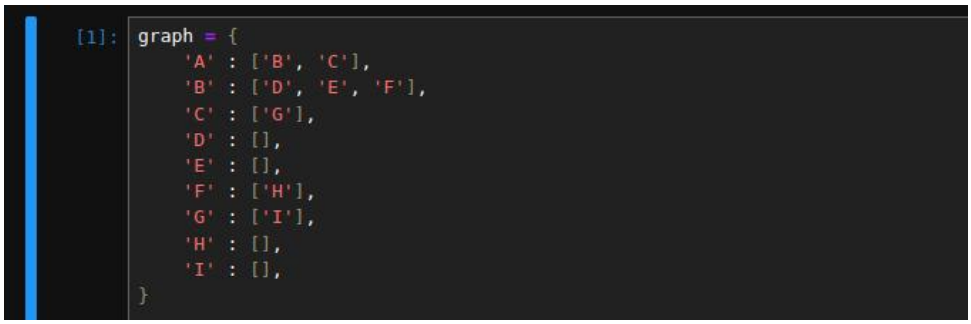
1. Pencarian Tidak Terarah dengan Depth-First Search

a) Kode Program



```
[1]: graph = {
         'A' : ['B', 'C'],
         'B' : ['D', 'E', 'F'],
         'C' : ['G'],
         'D' : [],
         'E' : [],
         'F' : ['H'],
         'G' : ['I'],
         'H' : [],
         'I' : [],
     }

     def dfs(graph, node):
         visited = []
         stack = []

         visited.append(node)
         stack.append(node)

         while stack:
             s = stack.pop()
             print(s, end = " ")

             for n in reversed(graph[s]):
                 if n not in visited:
                     visited.append(n)
                     stack.append(n)

     def main():
         dfs(graph,'A')

     main()

     A B D E F H C G I
```

- Graph



```
[1]: graph = {
         'A' : ['B', 'C'],
         'B' : ['D', 'E', 'F'],
         'C' : ['G'],
         'D' : [],
         'E' : [],
         'F' : ['H'],
         'G' : ['I'],
         'H' : [],
         'I' : [],
     }
```

Graph = {} adalah array untuk menampung informasi root dan child

- Fungsi dfs

```
def dfs(graph, node):
```

Mendefinisikan fungsi dfs (graph, node) yg dijalankan dengan variabel dari array graph dan node awal pencarian

Parameter:

- graph : Graf yang akan dicari.
- Node : Simpul awal pencarian.

- Visited dan Stack

```
    visited = []
    stack = []
```

Visited = [] sebagai array untuk menampung node yang telah dikunjungi
Stack = [] sebagai array untuk menampung node yang akan dikunjungi

- Append

```
        visited.append(node)
        stack.append(node)
```

Append untuk menambahkan isi pada list. Append adalah sebuah operasi yang digunakan untuk menambahkan satu atau lebih elemen ke akhir suatu struktur data.

- Pop

```
    while stack:
        s = stack.pop()
        print(s, end = " ")
```

Pop untuk mengeluarkan isi pada list yaitu node yang telah dikunjungi. Pop adalah operasi yang digunakan untuk menghapus dan mengembalikan elemen terakhir dari suatu struktur data.

- Pencarian

```
            for n in reversed(graph[s]):
                if n not in visited:
                    visited.append(n)
                    stack.append(n)
```

Jika ada node yang belum dikunjungi maka akan dimasukkan pada visited dan stack

- Menjalankan fungsi dfs() pada main function

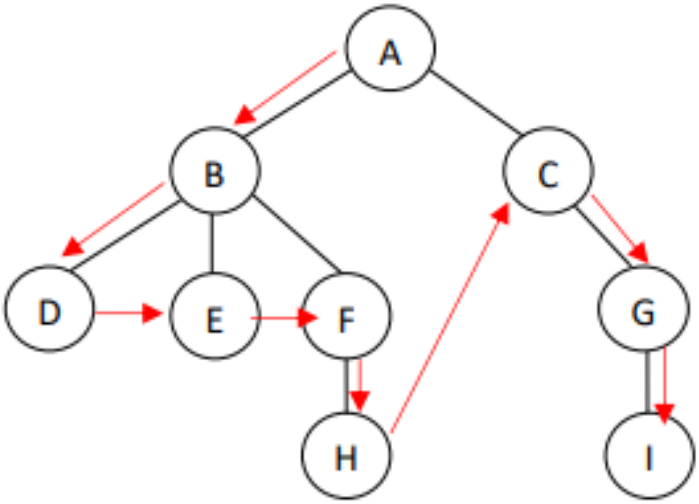
```
def main():
    dfs(graph,'A')

main()
```

b) Hasil

```
def main():
    dfs(graph,'A')

main()
A B D E F H C G I
```

Urutan ‘ABDEFHCGI’ menunjukkan bahwa algoritma DFS memulai pencarian dari simpul 'A', kemudian mengunjungi anak-anaknya secara mendalam ('B' lalu 'D'), sebelum kembali ke 'A' dan mengunjungi anak lainnya ('C'). Proses ini berlanjut hingga semua simpul yang terhubung dari 'A' telah dikunjungi.

c) Ilustrasi Pencarian



2. Pencarian Tidak Terarah dengan Breadth-First Search

a) Diagram

```
[2]: from collections import deque

graph = {
    'A' : ['B', 'C'],
    'B' : ['D', 'E', 'F'],
    'C' : ['G'],
    'D' : [],
    'E' : [],
    'F' : ['H'],
    'G' : ['I'],
    'H' : [],
    'I' : [],
}

def bfs(graph, node):
    visited = []
    queue = deque()

    visited.append(node)
    queue.append(node)

    while queue:

        s = queue.popleft()
        print(s, end = " ")

        for n in graph[s]:
            if n not in visited:
                visited.append(n)
                queue.append(n)

def main():
    bfs(graph,'A')

main()

A B C D E F G H I
```

- Import Deque

```
[2]: from collections import deque
```

Modul collections diimpor untuk menggunakan struktur data `deque` yang efisien untuk mengimplementasikan antrian.

- Fungsi BFS

```
def bfs(graph, node):
```

Fungsi ini melakukan pencarian BFS.

- Deque

```
queue = deque()
```

Deque (double-ended queue) adalah struktur data yang memungkinkan penambahan dan penghapusan elemen dari kedua ujungnya: ujung depan (head) dan ujung belakang (tail). Ini memberikan lebih banyak fleksibilitas dibandingkan dengan queue biasa yang hanya memungkinkan operasi pada ujung depan.

Operasi Dasar pada Deque

  - Append (or push_back): Menambahkan elemen ke ujung belakang.
  - Pop (or pop_back): Menghapus dan mengembalikan elemen dari ujung belakang.
  - Appendleft (or push_front): Menambahkan elemen ke ujung depan.
  - Popleft (or popleft): Menghapus dan mengembalikan elemen dari ujung depan.

- Popleft

```
while queue:

    s = queue.popleft()
    print(s, end = " ")
```

Queue.popleft() untuk mengeluarkan isi dari queue

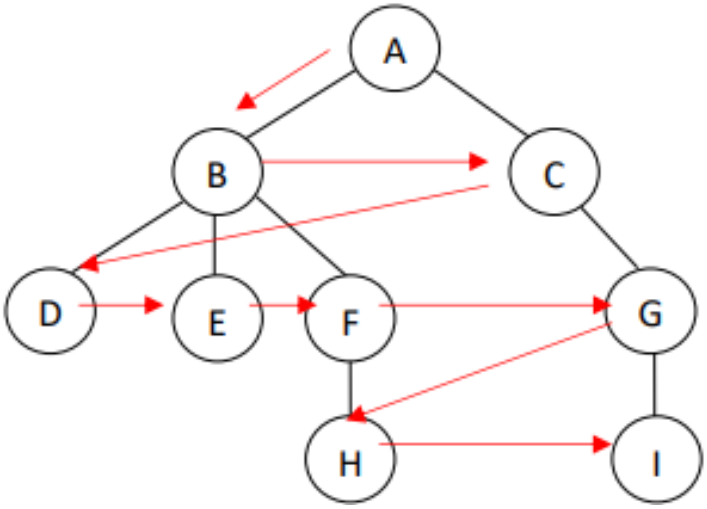
b) Hasil

```
A B C D E F G H I
```

Output dari kode tersebut adalah urutan kunjungan simpul pada graf, yaitu ABCDEFGHI. Ini berarti algoritma BFS mulai dari simpul 'A', kemudian mengunjungi semua tetangganya secara berlevel (level by level), lalu melanjutkan ke tetangga dari tetangga tersebut, dan seterusnya hingga semua simpul yang dapat dijangkau dari 'A' telah dikunjungi.

Urutan kunjungan BFS mencerminkan sifat level-by-level dari algoritma ini. BFS selalu mengeksplorasi semua simpul pada level saat ini sebelum pindah ke level berikutnya. Ini memastikan bahwa jalur terpendek dari simpul awal ke semua simpul lain dalam graf tak berbobot akan ditemukan.

c) Ilustrasi Pencarian



D. Kendala Praktikum

Tidak terdapat kendala pada praktikum ini

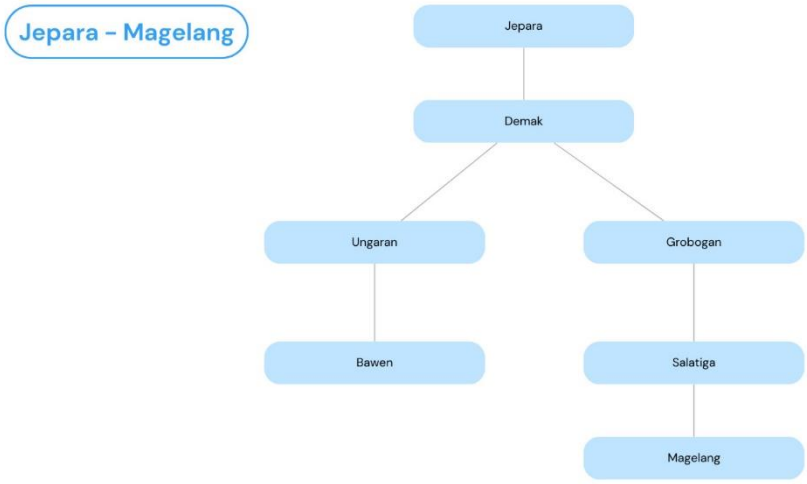
E. Studi Kasus

Implementasikan pencarian DFS pada kasus berikut:

Membuat kode program untuk menunjukkan alur dari Jepara menuju:

- Magelang
- Tegal
- Purwokerto

1. Alur Jepara – Magelang
   a) Diagram



   b) Kode Program

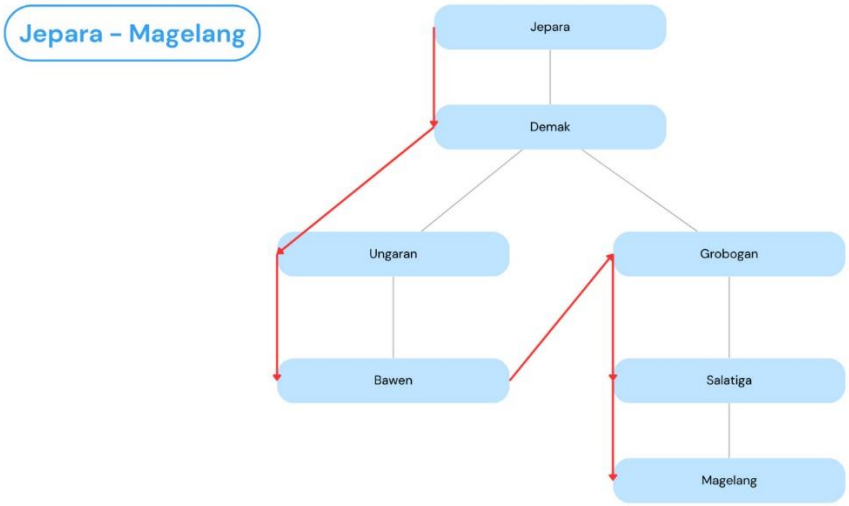
```
[5]: graph = {
        'Jepara' : ['Demak'],
        'Demak' : ['Ungaran', 'Grobogan'],
        'Ungaran' : ['Bawen'],
        'Bawen' : [],
        'Grobogan' : ['Salatiga'],
        'Salatiga' : ['Magelang'],
        'Magelang' : []
    }

    def dfs(graph, node):
        visited = []
        stack = []

        visited.append(node)
        stack.append(node)

        while stack:
            s = stack.pop()
            print(s, end = " ")

            for n in reversed(graph[s]):
                if n not in visited:
                    visited.append(n)
                    stack.append(n)


    def main():
        dfs(graph,'Jepara')

    main()
    Jepara Demak Ungaran Bawen Grobogan Salatiga Magelang
```

   c) Output

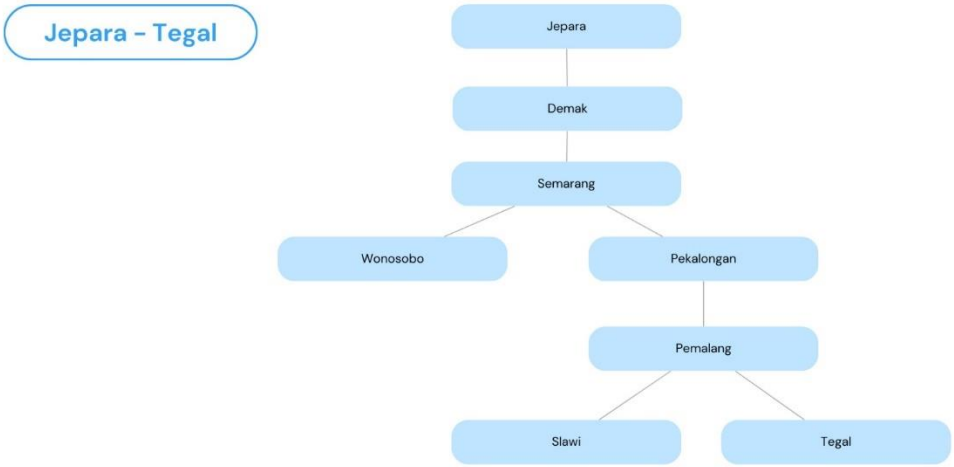
```
Jepara Demak Ungaran Bawen Grobogan Salatiga Magelang
```

Hasil dari pencarian DFS dengan titik awal 'Jepara' dan tujuan ‘Magelang’ adalah sebagai berikut: Jepara, Demak, Ungaran, Bawen, Grobogan, Salatiga, dan Magelang.

d) Ilustrasi Pencarian



2. Alur Jepara – Tegal
   a) Diagram



b) Kode Program

```
[8]: graph = {
        'Jepara' : ['Demak'],
        'Demak' : ['Semarang'],
        'Semarang' : ['Wonosobo','Pekalongan'],
        'Wonosobo' : [],
        'Pekalongan' : ['Pemalang'],
        'Pemalang' : ['Slawi','Tegal'],
        'Slawi' : [],
        'Tegal' : []
    }

    def dfs(graph, node):
        visited = []
        stack = []

        visited.append(node)
        stack.append(node)

        while stack:
            s = stack.pop()
            print(s, end = " ")
```

```
        for n in reversed(graph[s]):
            if n not in visited:
                visited.append(n)
                stack.append(n)

def main():
    dfs(graph,'Jepara')

main()
```

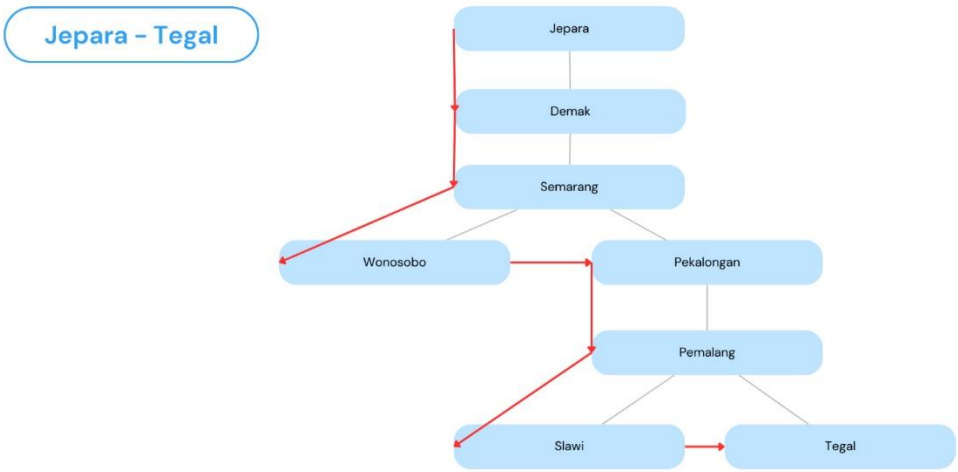
```
Jepara Demak Semarang Wonosobo Pekalongan Pemalang Slawi Tegal
```

c) Output

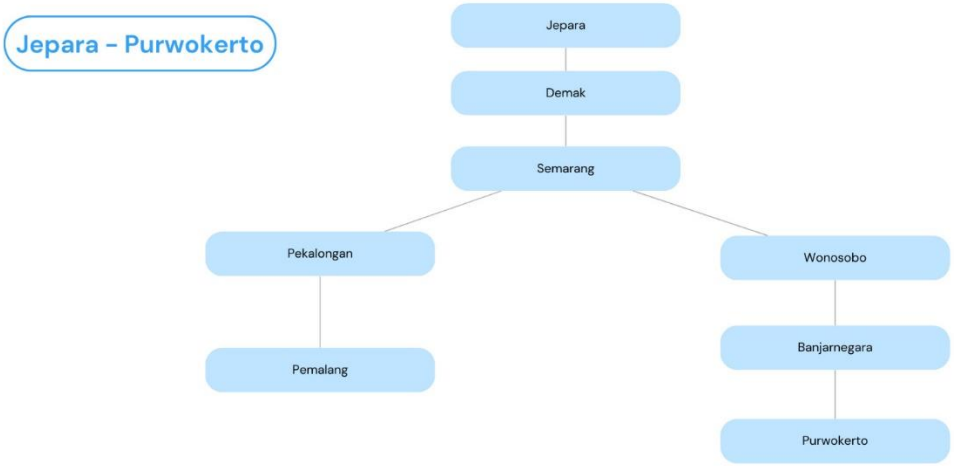
```
Jepara Demak Semarang Wonosobo Pekalongan Pemalang Slawi Tegal
```

Hasil dari pencarian DFS dengan titik awal 'Jepara' dan tujuan ‘Tegal’ adalah urutan kunjungan kota sebagai berikut: Jepara, Demak, Semarang, Wonosobo, Pekalongan, Pemalang, Slawi, dan Tegal.

d) Ilustrasi Pencarian



3. Alur Jepara – Purwokerto
   a) Diagram



b) Kode Program

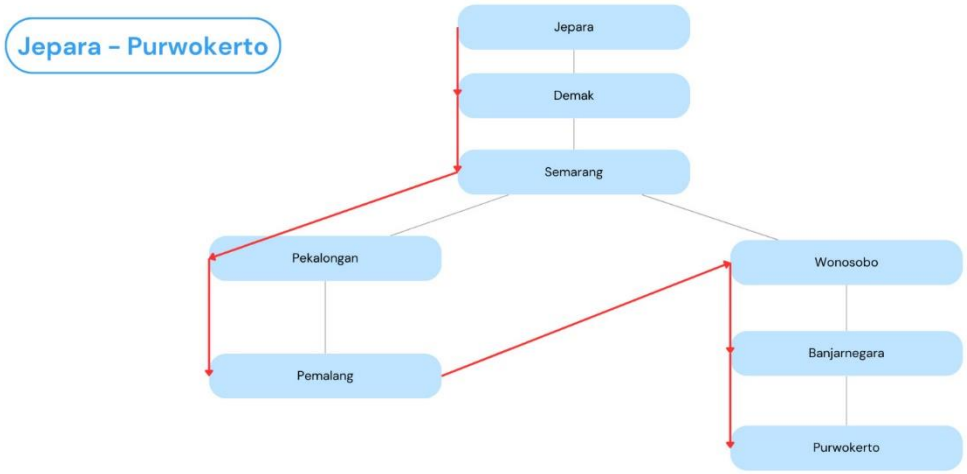
```
[21]: graph = {
          'Jepara' : ['Demak'],
          'Demak' : ['Semarang'],
          'Semarang' : ['Pekalongan', 'Wonosobo'],
          'Pekalongan' : ['Pemalang'],
          'Pemalang' : [],
          'Wonosobo' : ['Banjarnegara'],
          'Banjarnegara' : ['Purwokerto'],
          'Purwokerto' : []
      }

      def dfs(graph, node):
          visited = []
          stack = []

          visited.append(node)
          stack.append(node)

          while stack:
              s = stack.pop()
              print(s, end = " ")

              for n in reversed(graph[s]):
                  if n not in visited:
                      visited.append(n)
                      stack.append(n)

      def main():
          dfs(graph,'Jepara')

      main()
      Jepara Demak Semarang Pekalongan Pemalang Wonosobo Banjarnegara Purwokerto
```

c) Output

```
Jepara Demak Semarang Pekalongan Pemalang Wonosobo Banjarnegara Purwokerto
```

Hasil dari pencarian DFS dengan titik awal 'Jepara' dan tujuan ‘Purwokerto’ adalah sebagai berikut: Jepara, Demak, Semarang, Pekalongan, Pemalang, Wonosobo, Banjarnegara, dan Purwokerto.

d) Ilustrasi Pencarian



4. Kesimpulan

Dari ketiga studi kasus yang telah dilakukan, didapatkan bahwa hasil pencarian DFS berhasil menemukan rute dari Jepara ke Magelang, Jepara ke Tegal, dan Jepara ke Purwokerto. Tetapi algoritma DFS tidak menemukan jalur terpendek. Jika tujuan

utama adalah menemukan jalur terpendek, maka algoritma BFS lebih cocok digunakan.

Hal ini dikarenakan DFS cenderung menjelajahi satu cabang (jalur) sejauh mungkin sebelum kembali dan menjelajahi cabang lainnya. Hal ini berarti DFS akan menemukan jalur yang lebih panjang sebelum menemukan jalur yang lebih pendek. Sementara BFS menjelajahi semua simpul pada level yang sama sebelum pindah ke level berikutnya. Hal ini memastikan bahwa simpul yang lebih dekat dengan simpul awal akan dikunjungi terlebih dahulu.