Pembuatan Permainan serta Solusi Otomatis pada *Puzzle Tower of Hanoi* dengan Algoritma BFS

Fajar Maulana Herawan - 13521080[1]
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
[1]13521080@std.stei.itb.ac.id

***Abstract*—Dalam dunia teori bilangan serta kombinatorial pada bidang matematika dan informatika, *Tower of Hanoi* menjadi salah satu topik yang seringkali menjadi pokok pembahasan. *Tower of Hanoi* merupakan salah satu contoh permainan klasik tradisional yang menerapkan algoritma rekursif. Pada makalah ini, akan direalisasikan metode otomatisasi solusi *Tower of Hanoi* menggunakan salah satu konsep strategi algoritma, yaitu Algoritma *Breadth-First Search* (BFS). Permainan *Tower of Hanoi* tersusun atas tiga tiang dan beberapa variasi jumlah disk dengan ukuran yang berbeda-beda. Algoritma BFS diterapkan untuk melakukan eksplorasi terhadap keseluruhan solusi yang sistematik, serta dapat diterapkan pada penyelesaian *Tower of Hanoi*. Objektif dari penerapan ini adalah memastikan solusi optimal untuk tiap susunan *Tower of Hanoi*. Akan tetapi, dalam melakukan sebuah penyelesaian permainan *Tower of Hanoi*, algoritma BFS memerlukan sebuah komponen untuk melakukan operasi rekayasa tertentu. Pada makalah ini, pendekatan BFS akan direalisasikan dengan menggunakan suatu proses *enqueue* dan *dequeue* pada *queue*. Makalah ini akan berfokus untuk menerapkan algoritma rekursif dengan *Breadth-First Search* (BFS) serta keseluruhan aspek lainnya dalam pembuatan permainan sekaligus otomatisasi solusi *Tower of Hanoi*.**

Keywords—BFS; Hanoi; Otomisasi; Teori Bilangan; Queue;

I. PENDAHULUAN

Tower of Hanoi atau yang biasa disebut sebagai *The Problem of Benares Temple*, *Tower of Brahma*, *Lucas Tower*, dan beberapa nama lainnya merupakan sebuah permainan *puzzle* matematika yang terdiri atas tiga tiang serta kumpulan *disk*/piringan dengan ukuran diameter bervariasi. *Puzzle* dimulai dengan kumpulan piringan yang telah disusun pada suatu *disk* dengan urutan menurun, piringan dengan diameter terkecil berada pada bagian paling atas, dilanjutkan dengan piringan-piringan berdiameter lebih besar di bawahnya. Susunan yang dibentuk oleh piringan-piringan tersebut seolah-olah membentuk sebuah kerucut. Gambar 1.1 menunjukkan ilustrasi awal permainan *Tower of Hanoi* yang susunan awalnya berada pada tiang paling kiri. Pada permainan dengan 3 piringan, *puzzle* akan dapat diselesaikan dengan 7 pergerakan. Objektif dari permainan ini adalah untuk memindahkan keseluruhan piringan yang berada pada tiang paling kiri berpindah posisi pada tiang paling kanan. Proses pemindahan minimum untuk menyelesaikan permainan *Tower of Hanoi* adalah mengikuti persamaan 2^n-1, dengan *n* merupakan jumlah piringan yang digunakan. Akan tetapi, terdapat beberapa aturan yang harus ditaati dalam melakukan penyusunannya, antara lain:

1. Setiap pergerakan hanya dapat dilakukan dengan memindahkan satu buah piringan dari tiang awal ke tiang akhir.
2. Piringan yang dipindahkan merupakan piringan yang berada di posisi paling atas dalam sebuah susunan pada tiang tertentu.
3. Untuk setiap susunan, tidak diperbolehkan untuk memposisikan piringan dengan ukuran diameter tertentu berada di atas piringan berdiameter lebih kecil.

***Gambar 1.1** Ilustrasi Tower of Hanoi*
Sumber : https://www.etsy.com/listing/ 809045844/tower-of-hanoi-9-stem-puzzle-stem-math?frs=1

Permainan *Tower of Hanoi* pertama kali ditemukan oleh seorang ahli matematika asal Prancis bernama Edouard Lucas pada tahun 1883. Sejarah awal munculnya permainan ini masih menjadi pembicaraan yang tidak dapat dipastikan kebenarannya, termasuk salah satunya adalah mitos terkait Candi bercorak India di Kashi Vishwanath yang memiliki ruangan luas dengan tiga tiang besar di dalamnya serta dikelilingi oleh 64 piringan emas. Kisah tersebut pertama kali dipopulerkan oleh teman dari Eduoard Lucas.

Namun, kebenaran dari cerita tersebut masih dipertanyakan hingga saat ini. Apabila kisah tersebut benar serta pendeta pada zaman tersebut mampu memindahkan tiap piringan emas dari satu tiang menuju tiang lain dalam waktu satu detik, dengan jumlah pergerakan seminimal mungkin, maka proses penyelesaian permainan akan menghabiskan $2^{64}-1$ *detik* atau kurang lebih *585 juta tahun*.

Rentang waktu penyelesaian yang cukup panjang tersebut tentunya tidak mungkin diselesaikan oleh ahli pada zaman itu.

Akan tetapi, dengan menggunakan kemampuan komputer modern yang mampu melakukan komputasi sederhana dalam skala besar, maka penyelesaian dari permasalahan *Tower of Hanoi* dapat lebih mudah untuk disimulasikan. Permainan *puzzle Tower of Hanoi* merupakan salah satu contoh permainan kombinatorial yang dalam skala tertentu, penyelesaiannya masih dapat dilakukan menggunakan komputasi modern.

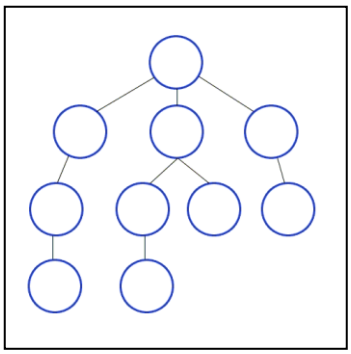
Permainan kombinatorial merupakan salah satu cabang ilmu kombinatorika yang mempelajari permainan dengan *perfect information* atau dengan kata lain seluruh pemain memiliki informasi lengkap mengenai kondisi dan keadaan permainan. Selain itu, permainan kombinatorial juga bersifat deterministik, yang berarti bahwa keseluruhan langkah akan memiliki konsekuensi yang pasti. Hal ini mengindikasikan bahwa permainan yang melibatkan lemparan dadu atau faktor keberuntungan tidak termasuk dalam permainan kombinatorial. Salah satu cabang permainan kombinatorial dimana permainan hanya dimainkan oleh satu orang disebut dengan *combinatorial puzzle*.

Dalam makalah ini, penulis akan lebih fokus terhadap pembahasan algoritma penyelesaian *Tower of Hanoi* yang cukup efektif dan efisien. Penulis akan menggunakan penerapan algoritma *Breadth-First Search* (BFS) dalam melakukan proses penyelesaian permainan *Tower of Hanoi*. Algoritma tersebut dipilih karena penerapannya yang cukup mudah, tingkat akurasi yang cukup, serta kesesuaiannya dengan materi pada mata kuliah IF2211 – Strategi Algoritma.

II. Landasan Teori

A. *Graf Traversal*

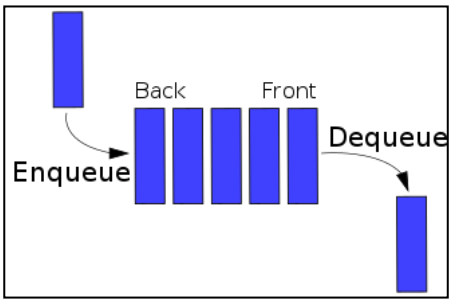
Definisi Graf Traversal adalah sebuah konsep dalam ilmu komputer untuk mengunjungi setiap node dari suatu graf secara traversal. Pada graf traversal, akan dilakukan pula proses menghitung urutan simpul dengan menggunakan teknik *traverse*. Secara umum, proses mengunjungi setiap node akan dimulai dari starting node (node awal) dan berakhir pada goal yang telah ditentukan sebelumnya. Setiap nodes yang telah dikunjungi akan disimpan informasi-nya agar tidak terjadi sebuah kondisi *infinite loop* akibat dari suatu node dikunjungi berulang-ulang kali. Proses pencatatan *visited node* dapat direkayasa menggunakan notasi biner, dengan memberikan nilai nol pada vertex yang telah dikunjungi dan satu pada vertex yang belum dikunjungi.



***Gambar 2.1** Ilustrasi Graf Traversal*
Sumber : https://www.thedshandbook.com/breadth-first-search/

B. *Queue*

Dalam bidang ilmu komputer, Queue merupakan kumpulan koleksi dari suatu entitas yang digunakan untuk menyimpan sebuah urutan tertentu dengan aturan penambahan elemen-nya berada pada bagian belakang dari urutan, sementara penghapusan elemen akan dilakukan pada elemen pertama dari urutan. Istilah yang seringkali digunakan dalam queue adalah Head dan Tail. Head mewakili elemen pertama atau elemen yang berada pada urutan paling awal dari suatu queue, sedangkan Tail mewakili elemen terakhir atau elemen yang berada pada urutan paling akhir dari suatu queue. Selain itu terdapat pula istilah yang berkaitan dengan proses penambahan maupun penghapusan pada queue. Proses penambahan elemen pada suatu queue sering disebut sebagai `*enqueue*`, sementara penghapusan suatu elemen pada queue disebut sebagai `*dequeue*`.



***Gambar 2.2** Ilustrasi Queue*
Sumber : https://developer-interview.com/p/algorithms/explain-principles-of-fifo-queue-and-lifo-stack-in-data-structures-47

Konsep penambahan dan penghapusan elemen pada queue seringkali lebih dikenal sebagai struktur data yang menerapkan prinsip *First-In-First-Out* (FIFO). Pada struktur data FIFO, elemen pertama yang ditambahkan pada queue akan pertama kali dihapus ketika operasi `*dequeue*` dilakukan. Hal ini tentu akan sejalan dengan aturan bahwa ketika suatu elemen baru ditambahkan, maka seluruh elemen yang telah ditambahkan sebelumnya harus dihapus terlebih dahulu sebelum elemen baru dapat dihapus. Oleh karena itu, prinsip *First-In-First-Out* (FIFO) seringkali disebut sebagai bentuk realisasi dari aturan antrian kehidupan nyata.

Selain operasi penambahan dan penghapusan elemen, terdapat pula beberapa operasi-operasi lainnya:

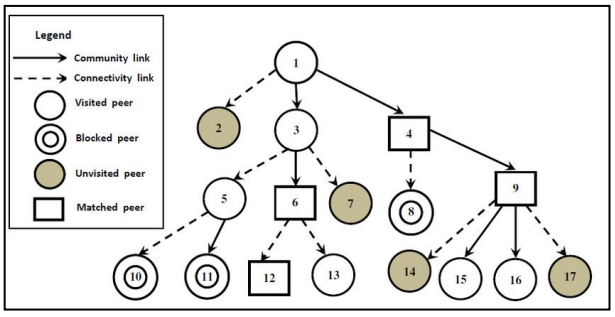
1. `*front*` : Operasi ini akan mengembalikan nilai dari elemen pertama suatu queue tanpa menghapusnya.
2. `*rear*` : Operasi ini akan mengembalikan nilai dari elemen terakhir suatu queue tanpa menghapusnya.
3. `*isEmpty*` : Operasi yang akan mengembalikan variabel boolean sebuah queue masih kosong atau tidak.
4. `*isFull*` : Operasi yang akan mengembalikan variabel boolean sebuah queue telah penuh atau belum.
5. `*size*` : Operasi yang akan mengembalikan ukuran dari sebuah queue (banyak elemen dalam sebuah queue).

Struktur data queue juga memiliki beberapa variasi dalam realisasinya. Variasi-variasi ini akan berkaitan dengan proses `*enqueue*` dan `*dequeue*` serta beberapa proses terkait lainnya. Beberapa variasi tipe dari queue, antara lain:

1. *Simple Queue* : Variasi *simple queue* atau yang biasa disebut sebagai *linear queue* merupakan variasi paling dasar dari suatu struktur data queue. Pada variasi ini, penambahan elemen akan dilakukan dengan operasi `*enqueue*` terhadap elemen terakhir, sementara penghapusan elemen akan dilakukan dengan operasi `*dequeue*` terhadap elemen pertama queue.
2. *Circular Queue* : Variasi *circular queue* merupakan variasi queue yang melakukan rekayasa terhadap elemen-elemen penyusunnya menjadi sebuah *circular ring*. Cara kerja dari *circular queue* sama seperti *simple queue*, hanya saja terdapat penambahan konsep bahwa elemen pertama dan terakhir dari queue saling terhubung. Keuntungan dari penerapan konsep ini adalah pada penghematan memori dengan menggunakan penempata posisi melalui operasi modulo.
3. *Priority Queue* : Variasi *priority queue* merupakan tipe spesial dari sebuah queue. *Priority queue* akan menyusun elemen-elemen dalam queue berdasarkan piroritas tertentu. Prioritas tersebut dapat didasarkan pada apapun, misalnya : dengan menggunakan prioritas elemen tertinggi, maka queue akan berada pada susunan *descending*.
4. *Dequeue* : Variasi *dequeue* atau *double ended queue*, merupakan variasi queue dimana operasi penambahan dan penghapusan elemen dapat dilakukan baik pada elemen pertama maupun elemen terakhir dari suatu queue. Akan tetapi, properti variasi dequeue sebenarnya telah melanggar konsep *First-In-First-Out* (FIFO) pada queue.

C. Breadth-First-Search (BFS)

Algoritma *Breadth-First Search* (BFS) merupakan salah satu algoritma yang paling sederhana dalam melakukan pencarian pada graf. Algoritma-algoritma pencarian pada graf lainnya, seperti algoritma prim pada permasalahan *minimum-spanning-tree*, algoritma djisktra pada pencarian solusi jarak terdekat, dan beberapa algoritma lainnya merupakan bentuk perluasan dari algoritma BFS. Dalam algoritma BFS, graf direpresentasikan dalam bentuk *G = (V, E)* dengan *v* merupakan simpul awal penelusuran.



***Gambar 2.3** Ilustrasi Breadth-First-Search*
Sumber : https://www.researchgate.net/figure/breadth-first-search-

Secara sederhana, algoritma *breadth-first search* akan memulai penelusuran dari simpul v. Kemudian akan dilakukan penelusuran terhadap seluruh simpul yang bertetanggan dengan simpul v terlebih dahulu. Terakhir, proses penelusuran dilanjutkan pada simpul-simpul lain yang belum dikunjungi dan bertetangga dengan simpul-simpul yang telah dikunjungi sebelumnya. Langkah-langkah tersebut akan dilakukan secara berulang-ulang hingga ditemukan simpul goal (simpul yang dicari) atau seluruh simpul telah ditelusuri.

Prosedur BFS dapat direalisasikan dengan menerapkan *adjacency lists* dalam merepresentasikan graf *G = (V, E)*. Selain itu untuk mengetahui urutan pemeriksaan suatu simpul maka akan digunakan struktur data queue. Terakhir, untuk mengetahui *unvisited* dan *visited node*, maka akan digunakan sebuah struktur data array atau *hash table* yang bertipe boolean. Berikut adalah pseudocode umum untuk prosedur *breadth-first-search* (BFS).

```
procedure BFS(input G: graph, input v: vertice)
{ Traversal Graf dengan algoritma penelususran BFS
  I.S. G dan v terdefinisi dan tidak sembarang
  F.S. Semua simpul yang dilalui tercetak pada layar }

KAMUS
  { Variabel }
  q : queue
  w : vertice
  visited : hashTable
  { Fungsi dan Prosedur antara }
  procedure createQueue(input/output q: queue)
  { Inisialisasi queue
    I.S. q belum terdefinisi
    F.S. q terdefinisi dan kosong }
  procedure enqueue(input/output q: queue, input v: vertice)
  { Memasukkan v ke dalam q dengan aturan FIFO
    I.S. q terdefinisi
    F.S. v masuk ke dalam q dengan aturan FIFO }
  function dequeue(q: queue) → vertice
  { Menghapus simpul dari q dengan aturan FIFO
    dan mengembalikan simpul yang dihapus}
  function isQueueEmpty(q: queue) → boolean
  { Mengembalikan True jika q kosong dan sebaliknya }
  procedure createHashTable(input/output T: hashTable)
  { Inisialisasi hashTable
    I.S. T belum terdefinisi
    F.S. T terdefinisi dan terisi False sebanyak simpul }
   procedure setHashTable(input/output T: hashTable, input v:
vertice, input val: boolean )
  { Mengubah value dari T
    I.S. T sudah terdefinisi
    F.S. Elemen T dengan key v sudah diubah menjadi val }
   function getHashTable(T: hashTable, input v: vertice) →
boolean
  { Mengembalikan elemen T pada key v }

ALGORITMA
  { Inisialisasi visited }
  createHashTable(visited)

  { Inisialisasi q }
  createQueue(q)

  { Mengunjungi simpul pertama (akar) }
  output(v)

  enqueue(q, v)
  setHashTable(T, v, True)

  { Mengunjungi semua simpul }
  while (not isQueueEmpty(q)) do
      v ← dequeue(q)
      for setiap simpul w yang bertetangga dengan simpul v do
          if (not getHashTable(visited, w)) then
              output(w)
              enqueue(q, w)
              setHashTable(T, w, True)

  { q kosong }
```

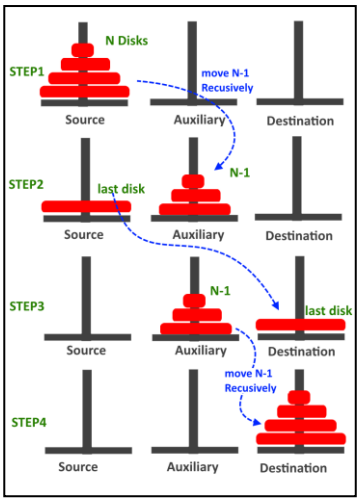
Pseudocode umum untuk prosedur *Breadth-First Search* di atas akan menghasilkan seluruh solusi berdasarkan input yang diberikan. Untuk pencarian satu simpul tertentu, algoritma dapat diubah menjadi:

```
procedure BFS(input G: graph, input v: vertice, input x: vertice)
{ Traversal Graf dengan algoritma penelususran BFS
   I.S. G dan v terdefinisi dan tidak sembarang
   F.S. Semua simpul yang dilalui tercetak pada layar }
KAMUS
  { Variabel }
  { IDEM }
  found : boolean
  { Fungsi dan Prosedur antara }
  { IDEM }

ALGORITMA
  { Inisialisasi found }
  found ← False
  { Inisialisasi visited }
  createHashTable(visited)
  { Inisialisasi q }
  createQueue(q)

  { Mengunjungi simpul pertama (akar) }
  output(v)
  enqueue(q, v)
  setHashTable(T, v, True)

  { Memeriksa simpul pertama (akar) }
  if (v = x) then
      found ← True

  { Mengunjungi semua simpul }
  while (not isQueueEmpty(q) and not found) do
      v ← dequeue(q)
      for setiap simpul w yang bertetangga dengan simpul v do
          if (not getHashTable(visited, w) and not found) then
          output(w)
          enqueue(q, w)
          setHashTable(T, w, True)
          if (w = x) then
              found ← True

  if (not found) then
      output("Tidak ditemukan")
```

D. *Solusi Tower of Hanoi*

Dalam melakukan penyelesaian permainan puzzle matematika *Tower of Hanoi* terdapat beberapa pendekatan yang dapat digunakan. Beberapa pendekatan tersebut antara lain:

1. Solusi Iteratif : Solusi sederhana untuk teka-teki permainan *Tower of Hanoi* adalah dengan mengganti gerakan antara potongan terkecil dan potongan non-terkecil. Saat melakukan pemindahan potongan terkecil, selalu pindahkan ke posisi berikutnya dengan arah yang konsisten (ke kanan jika jumlah potongan awal genap, ke kiri jika jumlah potongan awal ganjil). Jika tidak terdapat posisi tiang / menara pada arah yang dipilih, pindahkan bidak ke ujung yang berlawanan, tetapi lanjutkan dengan bergerak ke arah awal. Metode ini merupakan metode penyelesaian sederhana yang dapat digunakan untuk mencari solusi optimal permasalahan *Tower of Hanoi*.
2. Solusi rekursif : Solusi penyelesaian permasalahan rekursif dapat dilakukan ketika masalah tersebut dapat dibagi menjadi beberapa bagian sebagai sub-permasalahan.



Gambar 2.4 *Ilustrasi Solusi Tower of Hanoi*
Sumber : https://www.includehelp.com/data-structure-tutorial/tower-of-hanoi-using-recursion.aspx

Pada makalah ini, penyelesaian permainan *Tower of Hanoi* akan dilakukan menggunakan cara rekursif, sehingga algoritma *breadth-first search* dapat diterapkan dalam melakukan pencarian solusi permasalahan. Berikut ini adalah contoh penerapan algoritma yang digunakan dalam *python source code file*:

```python
A = [3, 2, 1]
B = []
C = []

def move(n, source, target, auxiliary):
    if n > 0:
        # Move n - 1 disks from source to auxiliary, so they are out of the way
        move(n - 1, source, auxiliary, target)

        # Move the nth disk from source to target
        target.append(source.pop())

        # Display our progress
        print(A, B, C, '###############', sep='\n')

        # Move the n - 1 disks that we left on auxiliary onto target
        move(n - 1, auxiliary, target, source)

# Initiate call from source A to target C with auxiliary B
move(3, A, C, B)
```

III. IMPLEMENTASI

A. *Struktur Data Program*

Directory Tree dari *source code* implementasi program adalah sebagai berikut:

```bash
```

```
    README.md
    requirements.txt

    ├──doc
    │   13521080-FajarMaulanaHerawan.doc
    │   13521080-FajarMaulanaHerawan.pdf
    │
    ├──img
    │   hanoi.gif
    │   icon.png
    │   SS1.png
    │   SS2.png
    │   SS3.png
    │
    └──src
        disk.py
        hanoi.py
```

Struktur *repository* program terdiri atas 3 folder utama serta 2 dokumen di luar folder, yaitu : *README.md* serta *requirements.txt*. `*requirements.txt*` berisi *library* atau pustaka yang diperlukan pada implementasi makalah ini. Sementara *README.md* merupakan dokumen yang berisi penjelasan utama *repository* ini. Folder pertama yang terdapat dalam repository ini adalah folder `*doc*` yang digunakan untuk menyimpan dokumen-dokumen terkait Tugas Makalah IF2211-Strategi Algoritma 2022/2023. Folder kedua merupakan folder `*img*` yang berisi gambar-gambar proyek ini. File yang berada dalam folder `*img*` antara lain: *SS1.png*, *SS2.png*, *SS2.png*, *icon.png*, serta *hanoi.gif*. *SS1.png* merupakan screenshot tampilan awal program, *SS2.png* merupakan screenshot pencarian solusi pada program, *SS3.png* merupakan screenshot tampilan akhir program, *icon.png* merupakan *favicon* yang digunakan ketika menjalankan program, serta *hanoi.gif* merupakan *video capture* ber-ekstensi *GIF* yang berisi tata cara penggunaan program.

Folder terakhir adalah folder `*src*` yang berisi 2 program utama dalam bentuk python (versi 3.9.4) *source code file*, bernama `*main.py*` dan `*disk.py*`. Pada *source code* tersebut akan dijalankan program `Permainan *Tower of Hanoi*`. Program akan ditampilkan menggunakan *library PyGame* sementara algoritma pencarian solusi akan diterapkan dengan algoritma *Breadth-First-Search* (BFS). Untuk lebih lengkapnya, *source code* dapat diakses melalui pranala berikut ini.

B. Implementasi Fungsi BFS

Pada makalah ini, fungsi pencarian solusi permainan *Tower of Hanoi* berada pada *source code* `*main.py*` yang terdapat dalam folder `*src*`. Fungsi penerapan pencarian solusi dalam kode program ini bernama `*preBFS*` dan `*BFS*`. Fungsi `*preBFS(p_s, p_f, pools)*`. Fungsi ini merupakan fungsi yang digunakan untuk mengeksekusi pemindahan kontrol sebelum menjalankan algoritma BFS. Fungsi ini menerima tiga parameter:

1. `*p_s*` adalah indeks pool asal (0, 1, atau 2) dari pemindahan kontrol.
2. `*p_f*` adalah indeks pool tujuan (0, 1, atau 2) dari pemindahan kontrol.
3. `*pools*` adalah representasi saat ini adri setiap pool dalam permainan.

Fungsi ini melakukan pemindahan kontrol pada `*pools*` sesuai dengan `*p_s*` dan `*p_f*`, kemudian mengembalikan hasilnya dalam `*new_pools*` dan status validitas pemindahan dalam `*valid*`. Berikut adalah kutipan kode program `*preBFS*` yang digunakan:

```python
# Mengeksekusi pemindahan kontrol sebelum menjalankan algoritma BFS
def preBFS(p_s, p_f, pools):
    new_pools = copy.deepcopy(pools)
    valid = False
    if len(new_pools[p_s]) > 0:
        if len(new_pools[p_f]) == 0:
            new_pools[p_f].append(new_pools[p_s][-1])
            new_pools[p_s] = new_pools[p_s][:-1]
            valid = True
        else:
            if new_pools[p_s][-1] < new_pools[p_f][-1]:
                new_pools[p_f].append(new_pools[p_s][-1])
                new_pools[p_s] = new_pools[p_s][:-1]
                valid = True
    return new_pools, valid
```

***Gambar 3.1** Ilustrasi Fungsi preBFS*
Sumber : Dokumen Pribadi

Fungsi selanjutnya yang diperlukan dalam implementasi algoritma BFS pada pencarian solusi permainan *Tower of Hanoi* adalah fungsi `*BFS(pool_stat)*`. Fungsi ini merupakan implementasi algoritma BFS untuk mencari solusi optimal dalam permainan *Tower of Hanoi*. Fungsi `BFS` menerima satu paramater bernama `*pool_stat*`, sebagai representasi saat ini dari setiap pool dalam permainan.

Alur kerja algoritma dimulai dengan dua kelompok: `*searching*` untuk menyimpan konfigurasi yang akan diperiksa dan `*searched*` untuk menyimpan konfigurasi yang telah diperiksa. Selanjutnya, algoritma melakukan perulangan hingga menemukan solusi optimal. Pada setiap iterasi, algoritma mengambil konfigurasi pertama dari `*searching*` dan memeriksa apakah jumlah disk pada salah satu pool tujuan (pool kedua atau pool ketiga) sudah mencapai jumlah total *disk* (N). Jika sudah, berarti solusi optimal telah ditemukan dan perulangan dihentikan.

Jika solusi belum ditemukan, algoritma melakukan pemindahan kontrol dari setiap pool asal (`*p_s*`) ke setiap pool tujuan (`*p_f*`) yang berbeda. Sebelum pemindahan dilakukan, terjadi pemeriksaan validitas dengan memanggil fungsi `*preBFS*`. Fungsi `*preBFS*` melakukan pemindahan kontrol pada konfigurasi saat ini dan mengembalikan konfigurasi yang baru (`*new_pools*`) serta status validitas pemindahan (`*valid*`).

Apabila pemindahan kontrol valid, konfigurasi baru serta riwayat perpindahan (`*hist*`) ditambahkan ke dalam `*searching*` dan `*searched*`. Hal ini memungkinkan algoritma untuk mengeksplorasi semua kemungkinan pemindahan secara sistematis sebelum melanjutkan ke konfigurasi berikutnya. Dengan demikian, algoritma ini akan mencari solusi dengan jumlah pemindahan minimum dan menghasilkan urutan langkah-langkah yang optimal untuk menyelesaikan permainan *Tower of Hanoi*. Setelah menemukan solusi optimal, algoritma mengembalikan riwayat perpindahan (`*hist*`) sebagai hasil.

Riwayat perpindahan tersebut merepresentasikan urutan langkah-langkah yang harus diambil untuk mencapai solusi optimal dalam permainan *Tower of Hanoi*. Dengan menggunakan algoritma BFS, kode tersebut dapat mencari dan menghasilkan solusi optimal dengan menjelajahi secara sistematis semua kemungkinan perpindahan serta meminimalkan jumlah pemindahan yang diperlukan untuk menyelesaikan permainan *Tower of Hanoi*. Berikut adalah kutipan kode program `*BFS*` yang digunakan:

```python
# Algoritma BFS untuk mencari solusi optimal
def BFS(pool_stat):
    searching = [(pool_stat, [])]
    searched = [pool_stat]

    searchedN = 0

    while True:
        searchedN += 1
        if searchedN % 1000 == 0:
            print('%s : Searching ... '%(searchedN))
        pool_stat, hist = searching[0]
        if len(pool_stat[2]) == N or len(pool_stat[1]) == N:
            break
        searching = searching[1:]
        for p_s in range(3):
            for p_f in range(3):
                if p_s != p_f:
                    new_pools, valid = preBFS(p_s, p_f, pool_stat)
                    if valid:
                        if new_pools not in searched:
                            searching.append((new_pools, hist+[(p_s, p_f)]))
                            searched.append(new_pools)
    return hist
```

Gambar 3.2 *Ilustrasi Fungsi BFS*
Sumber : Dokumen Pribadi

C. *Implementasi Fungsi Perantara*

Dalam implementasi program, terdapat tiga fungsi perantara, yaitu : *gameInit()*, *addControl(pool_ind)*, serta *listPool()*. Berikut ini adalah kutipan dari kode *gameInit()* yang digunakan:

```python
def gameInit():
    # Variabel global
    global pools, preBFS_N, start, finish, status, auto, optN

    # Daftar pool
    pools = []
    pools.append([])
    pools.append([])
    pools.append([])

    # Menambahkan disk ke pool pertama
    for i in range(N, 0, -1):
        pools[0].append(disk(i, (170 + round(85 * (1 - i / N)), 220,
170 + round(85 * i / N)), DISPLAYSURF, disk_d, disk_b))

    # Variabel pool awal dan akhir
    start = -1
    finish = -1

    # Jumlah langkah BFS
    preBFS_N = 0

    # Status permainan
    status = 0

    # Mode otomatis
    auto = False

    # Jumlah langkah optimal
    optN = 1
    for i in range(N - 1):
        optN = 2 * optN + 1
```

Gambar 3.3 *Ilustrasi Fungsi gameInit*
Sumber : Dokumen Pribadi

Fungsi *gameInit()* merupakan fungsi yang digunakan untuk menginisialisasi permainan. Alur kerja fungsi diawali dengan melakukan pendeklarasian beberapa variabel *global*, seperti `*pools*`, `*preBFS_N*`, `*star*`, `*finish*`, `*status*`, `*auto*`, dan `*optN*`. Selanjutnya sebuah list bernama `*pools*` dibuat untuk mewakili tiga pool dalam permainan *Tower of Hanoi*. Piringan-piringan dengan ukuran yang berbeda ditambahkan ke pool pertama `*pools[0]*`. Variabel `*start*` dan `*finish*` diatur sebagai -1 untuk menandakanan bahwa belum ada pool asal dan tujuan yang dipilih. Selain itu, jumlah langkah pada pencarian BFS (`*preBFS_N*`) diatur sebagai 0. Status permainan (`*status*`) diatur sebagai 0 untuk menandakan bahwa permainan belum dimulai. Mode otomatis (`*auto*`) diatur sebagai *False*. Serta jumlah langkah optimal (`*optN*`) dihitung berdasarkan jumlah disk (N) menggunakan rumus $2^N - 1$.

```python
# Fungsi untuk menambahkan kontrol ke dalam pool
def addControl(pool_ind):
    global start, finish, selected_disk, disk_x, disk_y, new_disk_x, new_disk_y, status

    if start == -1 and status == 0:
        # Menambahkan kontrol jika pool tidak kosong
        if len(pools[pool_ind]) > 0:
            start = pool_ind
            disk_x = (2*pool_ind + 1) * pool_W + 21
            disk_y = WINDOW_H - 40 - len(pools[pool_ind])*(disk_b + disk_d)
            selected_disk = pools[pool_ind][-1]
            pools[start] = pools[pool_ind][:-1]
            status = 1

    elif start >= 0 and (status == 1 or status == 2) :
        # Memindahkan kontrol ke pool lain
        if len(pools[pool_ind]) == 0:
            finish = pool_ind
            new_disk_x = (2*pool_ind + 1) * pool_W + 21
            new_disk_y = WINDOW_H - 40 - (len(pools[pool_ind]) + 1)*(disk_b + disk_d)
            if status == 2:
                status = 3
        else:
            if pools[pool_ind][-1].disk_ind > selected_disk.disk_ind:
                finish = pool_ind
                new_disk_x = (2*pool_ind + 1) * pool_W + 21
                new_disk_y = WINDOW_H - 40 - (len(pools[pool_ind]) + 1)*(disk_b + disk_d)
                if status == 2:
                    status = 3
```

Gambar 3.4 *Ilustrasi Fungsi addControl*
Sumber : Dokumen Pribadi

Fungsi perantara kedua adalah fungsi `*addControl(pool_ind)*`. Fungsi ini digunakan untuk menambahkan kotrol (piringan) ke dalam pool. Alur kerja dari fungsi ini diawali dengan pendeklarasian beberapa variabel global speerti `*start*`, `*finish*`, `*selected_disk*`, `*disk_x*`, `*disk_y*`, `*new_disk_x*`, dan `*new_disk_y*`. Jika belum terdapat pool asal yang dipilih (`*start==-1*`) dan status permainan (`*status*`) adalah 0, maka kontrol dapat ditambahkan ke pool yang dipilih jika pool tersebut tidak kosong. Apabila kontrol berhasil ditambahkan, variabel `*start*` diatur sebagai pool asal yang dipilih, dan variabel `*disk_x*` dan `*disk_y*` diatur sebagai posisi awal kontrol. Selanjutnya, kontrol tersebut dihapus dari pool awal (`*pools[start]*`), dan status permainan diubah menjadi 1 untuk menandakan pemilihan pool asal telah selesai.

Apabila sudah ada pool asal yang dipilih (`*start >= 0*`) dan status permainan adalah 1 atau 2, maka kontrol dapat dipindahkan ke pool tujuan. Jika pool tujuan kosong, variabel finish diatur sebagai pool tujuan yang dipilih, serta variabel `*new_disk_x*` dan `*new_disk_y*` diatur sebagai posisi baru kontrol. Jika pool tujuan tidak kosong, kontrol dapat dipindahkan hanya jika disk pada pool tujuan memiliki ukuran yang lebih besar dari disk yang dipilih. Apabila kontrol berhasil dipindahkan, variabel `*finish*` diatur sebagai pool

tujuan yang dipilih, serta variabel `*new_disk_x*` dan `*new_disk_y*` diatur sebagai posisi baru kontrol. Serta langkah terakhir, jika status permainan adalah 2, status diubah menjadi 3 untuk menandakan pemindahan kontrol sedang berlangsung. Sementara fungsi perantara tearkhir adalah `*listPoll()*` yang berguna untuk menghasilkan daftar pool saat ini. Berikut ini adalah kutipan implementasi dari fungsi perantara *listPool* tersebut:

```
# Menghasilkan daftar pool
def listPoll():
    pool_list = []
    for pool in pools:
        temp = []
        for disk in pool:
            temp.append(disk.disk_ind)
        pool_list.append(temp)
    return pool_list
```

Gambar 3.5 *Ilustrasi Fungsi listPoll*
Sumber : Dokumen Pribadi

D. Kelas Disk

Kelas `*disk*` pada folder `*src/disk.py*` digunakan untuk merepresentasikan sebuah piringan dalam permainan *Tower of Hanoi*. Pada kelas ini terdapat dua buah metode, yaitu:

1. Konstruktor `*__init__(self, disk_ind, color, disp, disk_d, disk_b)*` : konstruktor ini digunakan untuk menginisialisasi objek `*disk*` dengan parameter berikut:
    - `*disk_ind*`: indeks piringan yang menentukan ukuran piringan.
    - `*disp*` : tampilan dimana piringan akan digambar
    - `*diskd*` : lebar atau tinggi piringan
    - `*diskb*` : ketebalan piringan
2. Metode `*draw(self, x, y)*` : metode ini digunakan untuk menggambar disk pada tampilan game dengan posisi `*(x,y)*`.
    - Metode ini menggunakan fungsi `*pygame.draw.rect()*` untuk menggambar persegi panjang yang mewakili piringan.
    - Parameter pertama adalah tampilan `*self.disp*`.
    - Parameter kedua adalah warna piringan `*self.color*`.
    - Parameter ketiga adalah koordinat dan ukuran persegi panjang yang akan digambar, dihitung berdasarkan atribut-atribut disk seperti `*self.disk_w*`, `*self.disk_d*`, dan `*self.disk_b*`.
    - Posisi `*(x, y)*` digunakan untuk menentukan posisi tengah bawah persegi panjang yang akan digambar.

Dengan menggunakan kelas `*disk*`, setiap piringan dalam permainan *Tower of Hanoi* direpresentasikan sebagai objek piringan dengan ukuran, warna, dan metode yang sesuai untuk menggambar piringan tersebut pada tampilan yang ditentukan. Berikut ini adalah kutipan implementasi kelas `*disk*` yang digunakan:

```
class disk:
    def __init__(self, disk_ind, color, disp, disk_d, disk_b):
        self.disk_ind = disk_ind
        self.color = color

        self.disk_w = 20 + 25 * (disk_ind)
        self.disk_d = disk_d
        self.disk_b = disk_b

        self.disp = disp


    def draw(self, x, y):
        pygame.draw.rect(self.disp, self.color, (x - self.disk_w / 2, y, self.disk_w, self.disk_d))
```

Gambar 3.6 *Ilustrasi Kelas disk*
Sumber : Dokumen Pribadi

E. Pustaka PyGame

Pada makalah ini, visualisasi serta manipulasi hasil implementasi program diterapkan menggunakan *library PyGame* (versi 2.4.0) yang tersedia pada bahasa pemrograman Python. Beberapa fungsi yang digunakan pada kode program antara lain:

1. *pygame.init()*: Menginisialisasi modul *pygame*.
2. *pygame.display.set_icon(icon)*: Mengatur ikon game dengan menggunakan gambar yang telah dimuat menggunakan *pygame.image.load()*.
3. *pygame.display.set_caption("Tower of Hanoi Game")*: Mengatur judul jendela game.
4. *FPSCLOCK = pygame.time.Clock():* Membuat objek *clock* untuk mengontrol kecepatan frame per detik.
5. *DISPLAYSURF=pygame.display.set_mode((WINDOW_W, WINDOW_H))*: Membuat *surface* (permukaan) tampilan dengan ukuran yang ditentukan.
6. *pygame.draw.line(DISPLAYSURF, (200, 200, 200), ((2*p_i + 1) * pool_W + 20, WINDOW_H - pool_L - 40), ((2*p_i + 1) * pool_W + 20, WINDOW_H - 30), 6)*: Menggambar garis sebagai tampilan kolom pool.
7. *pygame.draw.line(DISPLAYSURF, (240, 240, 240), (20, WINDOW_H - 30), (WINDOW_W - 20, WINDOW_H - 30), 10)*: Menggambar garis sebagai tampilan dasar pool.
8. *pygame.draw.rect(self.disp, self.color, (x - self.disk_w / 2, y, self.disk_w, self.disk_d))*: Menggambar persegi panjang sebagai tampilan piringan.
9. *pygame.event.get()*: Mendapatkan daftar semua event yang terjadi pada *pygame*.
10. *pygame.quit()*: Keluar dari *pygame*.
11. *sys.exit():* Menghentikan program.
12. *pygame.mouse.get_pos()*: Mengembalikan posisi x dan y dari *mouse*.
13. *pygame.font.Font('freesansbold.ttf', 20):* Membuat objek *font* untuk digunakan dalam render teks.
14. *pygame.font.Font.render():* Merender teks dengan menggunakan font yang telah dibuat sebelumnya.
15. *DISPLAYSURF.blit():* Menggambar *surface* (permukaan) pada layar.
16. *pygame.display.update():* Memperbarui tampilan dengan menggambar *surface* pada layar.

17. *FPSCLOCK.tick(FPS)*: Mengatur kecepatan frame per detik dengan menggunakan objek *clock* yang telah dibuat sebelumnya.

F. Langkah Kerja Hasil Implementasi Program

Program memiliki beberapa fitur yang dapat digunakan oleh pengguna, antara lain:

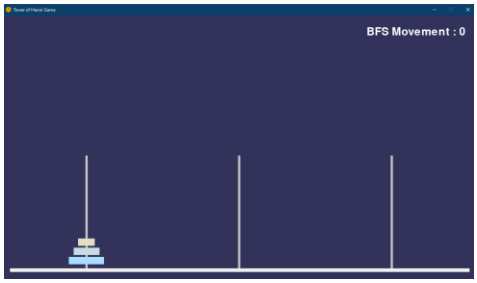
1. Fitur untuk memindahkah piringan secara manual
2. Fitur untuk mencari solusi otomatis dengan algoritma BFS
3. Fitur untuk menambah dan mengurangi jumlah piringan
4. Fitur untuk mempercepat dan memperlambat *game speed*

Fitur untuk memindahkan piringan secara manual dapat dilakukan dengan menekan `*left click*` pada *mouse* dan menekannya kembali pada pool tujuan. Fungsi untuk mencari solusi otomatis dengan algoritma *Breadth-First-Search* dapat dilakukan dengan menekan tombol `*s*` pada *keyboard*. Fitur untuk menambah dan mengurangi jumlah piringan dapat dilakukan dengan menekan tombol `*left/right arrow*` pada *keyboard*. Sementara fitur untuk mempercepat dan memperlambat *game speed* dapat dilakukan dengan menekan tombol `*up/down arrow*` pada *keyboard*.

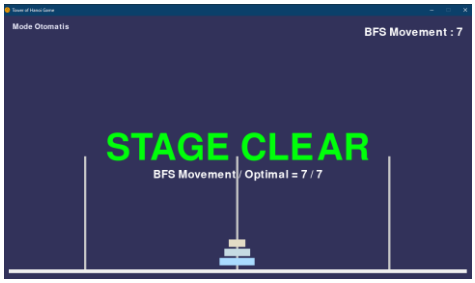
IV. PENGOLAHAN DATA DAN UJI COBA

A. Ujicoba Solusi Otomatis Tower of Hanoi

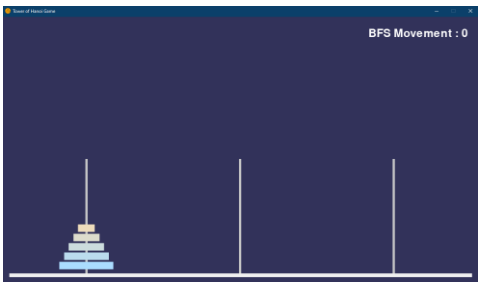
Pada makalah ini, proses pengujian akan dilakukan sebanyak 3 kali. Variasi Percobaan yang dilakukan adalah terkait jumlah piringan yang digunakan dalam permainan. Berikut ini adalah hasil percobaan yang dilakukan:



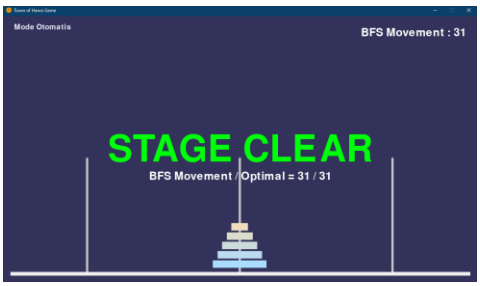
Gambar 4.1 *Ujicoba 3 Piringan*
Sumber : Dokumen Pribadi



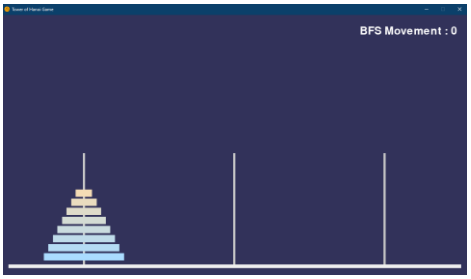
Gambar 4.2 *Hasil Ujicoba 3 Piringan*
Sumber : Dokumen Pribadi



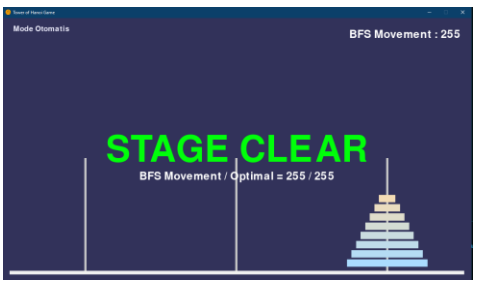
Gambar 4.3 *Ujicoba 5 Piringan*
Sumber : Dokumen Pribadi



Gambar 4.4 *Hasil Ujicoba 5 Piringan*
Sumber : Dokumen Pribadi



Gambar 4.5 *Ujicoba 8 Piringan*
Sumber : Dokumen Pribadi



Gambar 4.6 *Hasil Ujicoba 8 Piringan*
Sumber : Dokumen Pribadi

B. Pengolahan Data

Dari tiga kali ujicoba dan enam gambar di atas, telah dicatat jumlah solusi (langkah) yang dihasilkan untuk menyelesaikan tiap persoalan *Tower of Hanoi*. Variasi dilakukan dalam tiga variasi ujicoba, antara lain : tiga piringan, lima piringan, serta delapan piringan. Nilai hasil penyelesaian *tower of hanoi* dengan algoritma *Breadth-First-Search* inilah yang nantinya akan diuji dengan perhitungan menggunakan rumus. Dengan demikian, dapat diambil kesimpulan terkait tingkat efektif dan efisiensi dari kode program yang dibuat dalam proyek ini. Berikut ini adalah tabel hasil dari tiga percobaan di atas

(ditambah dengan 3 variansi percobaan lain yang gambar-nya tidak dilampirkan):

Tabel 4.1 *Data Jumlah Langkah*
Sumber : Dokumen Pribadi

No	Jumlah Disk	Langkah	Solusi Optimal 2^N -1	Ket
1	3	7	7	Optimal
2	4	15	15	Optimal
3	5	31	31	Optimal
4	6	63	63	Optimal
5	7	127	127	Optimal
6	8	255	225	Optimal

Dari enam percobaan yang dilakukan, seluruhnya berjumlah sama dengan hasil perhitungan menggunakan persamaan 2^N *-1*. Hal ini membuktikan bahwa perhitungan yang dilakukan telah efektif, efisien, serta optimal. Dengan demikian tujuan dari pembuatan makalah ini telah berhasil dicapai.

V. KESIMPULAN

Tower of Hanoi merupakan permainan tradisional klasik yang seringkali menjadi topik pembahasan dalam bidang informatika maupun matematika, terkhusus pada bidang teori bilangan dan kombinatorial. Objektif dari permainan ini adalah memindahkan keseluruhan piringan dari suatu tiang menuju tiang lain, tetapi dengan batasan dan aturan tertentu, Terdapat dua metode pendekatan yang lazim digunakan untuk menyelesaikan permasalahan *Tower of Hanoi*, yaitu : metode iteratif dan rekursif. Metode rekursif pada penyelesaian permainan *Tower of Hanoi* dapat dilakukan dengan menggunakan algoritma *Breadth-First Search* (BFS). Algoritma BFS nantinya akan dimulai dengan menyusun sebuah queue yang berisi urutan langkah yang hendak dilakukan dan tidak melanggar batasan sebelumnya. Pencarian akan terus dilakukan hingga didapatkan bahwa keseluruhan piringan telah berpindah pada satu tiang tujuan tertentu. Algoritma BFS dapat diimplementasikan untuk membuat sebuah "Pencarian Solusi Otomatis pada Permainan *Tower of Hanoi*" seperti yang dilakukan pada makalah ini. Penerapan konsep-konsep memanfaatkan beberapa pustaka untuk dapat mencapai tujuan makalah ini, antara lain : *PyGame*, *os*, dan lain sebagainya. "Pencarian Solusi Otomatis pada Permainan *Tower of Hanoi*" berhasil diimplementasikan dalam makalah ini, dibuktikan dengan data hasil ujicoba pada Bab 4. Selain itu, tujuan dari makalah ini berhasil dicapai, yaitu membuat penyelesaian permasalahan *Tower of Hanoi* yang optimal dengan algoritma BFS.

LINK VIDEO YOUTUBE

Link video youtube yang berisi penjelasan program, cara penggunaan program, serta pembahasan makalah dapat diakses melalui pranala berikut ini.

UCAPAN TERIMA KASIH

Puji dan syukur penulis panjatkan kepada Tuhan Yang Maha Esa atas segala rahmat dan kasih karunia-Nya yang telah memberikan kesehatan dan kesempatan kepada penulis sehingga penulis dapat menyelesaikan makalah ini. Penulis juga mengucapkan terima kasih sebesar-besarnya kepada seluruh pihak yang telah membantu penulis dalam menyelesaikan makalah ini, antara lain:

1. Dr. Nur Ulfa Maulidevi, S.T, M.Sc. sebagai dosen pengampu dalam mata kuliah IF2211 Strategi Algoritma K02.
2. Dr. Ir. Rinaldi Munir, M.T. atas kontribusi buku dan materi yang penulis kutip pada makalah ini.
3. Seluruh pihak yang telah membuat *source code* dan modul secara *open-source* serta penulis kutip pada makalah ini.

REFERENCES

[1] Rinaldi Munir. 2023. *Diktat Algoritma BFS dan DFS*. https://informatika.stei.itb.ac.id/~rinaldi.munir/Stmik/2020-2021/BFS-DFS-2021-Bag1.pdf, diakses pada 15 Mei 2023.

[2] IncludeHelp. 2017. *Tower of Hanoi Using Recursion*. https://www.includehelp.com/data-structure-tutorial/tower-of-hanoi-using-recursion.aspx, diakses pada 15 Mei 2023.

[3] Jaydip Sen. 2013. *An Efficient, Secure and User Privacy-Preserving Search Protocol for Peer-to-Peer Networks*. https://www.researchgate.net/figure/The-breadth-first-search-BFS-tree-for-the-search-initiated-by-peer-1_fig2_305381181, diakses pada 15 Mei 2023.

[4] The Shand Book. 2020. *Breadth First Search*. https://www.thedshandbook.com/breadth-first-search/, diakses pada 16 Mei 2023.

[5] Developer Interview. 2023. Explain principles of FIFO (queue) and LIFO (stack) in data structures. https://developer-interview.com/p/algorithms/explain-principles-of-fifo-queue-and-lifo-stack-in-data-structures-47, diakses pada 16 Mei 2023.

[6] GeeksForGeeks. 2023. Queue Data Structure. https://www.geeksforgeeks.org/queue-data-structure/, diakses pada 17 Mei 2023.

[7] GeeksForGeeks. 2023. Program for Tower of Hanoi Algorithm. https://www.geeksforgeeks.org/c-program-for-tower-of-hanoi/, diakses pada 17 Mei 2023.

[8] Paul Cull, E. F.Ecklund. Jr. 2018. Towers of Hanoi and Analysis of Algorithms.https://www.tandfonline.com/doi/abs/10.1080/00029890.1985.11971635, diakses pada 18 Mei 2023.

[9] Manfred Stadel. 1984. Another nonrecursive algorithm for the towers of Hanoi. https://dl.acm.org/doi/abs/10.1145/948596.948602, diakses pada 18 Mei 2023.

[10] Zhiming Liu, Anders P. Ravn. 2009. Automated Technology for Verification and Analysis. https://link.springer.com/book/10.1007/978-3-642-04761-9#page=192, diakses pada 18 Mei 2023.

[11] Andreas M. Hinz , Sandi Klavžar , Ciril Petr. 2018. The Tower of Hanoi – Myths and Maths. https://link.springer.com/book/10.1007/978-3-319-73779-9, diakses pada 19 Mei 2023.

[12] F.B. Chedid, T. Mogi. 1996. A simple iterative algorithm for the towers of Hanoi problem. https://ieeexplore.ieee.org/abstract/document/502075, diakses pada 19 Mei 2023.

[13] Mario Szegedy. 2002. In How Many Steps the k Peg Version of the Towers of Hanoi Game Can Be Solved?. https://link.springer.com/chapter/10.1007/3-540-49116-3_33, diakses pada 19 Mei 2023.

[14] John J.Paez. 2020. Use of Digital Tools to Promote Understanding of the Learning Process in the Tower of Hanoi Game. https://dl.acm.org/doi/abs/10.1145/3445945.3445950, diakses pada 19 Mei 2023.

[15] Rostislav Grigorchuk, Zoran Sunic. 2007. Schreier spectrum of the Hanoi Towers group on three pegs. https://arxiv.org/abs/0711.0068, diakses pada 19 Mei 2023.

[16] Xi Chen, Jingsai Liang. 2023. Teaching Graph Algorithms Using Tower of Hanoi and Its Variants. https://dl.acm.org/doi/abs/10.1145/3545947.3576351, diakses pada 19 Mei 2023.

[17] Richard E. Korf. 2000. Delayed Duplicate Detection: Extended Abstract. https://citeseerx.ist.psu.edu/document?repid=rep1&type=pdf&doi=83da3f4313f067d3c89e28b1191722166f605978, diakses pada 19 Mei 2023.

[18] Vinod Goel, Jordan Grafman. 1995. Are the frontal lobes implicated in "planning" functions? Interpreting data from the Tower of Hanoi. https://www.sciencedirect.com/science/article/abs/pii/002839329590866P, diakses pada 20 Mei 2023.

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, 20 Mei 2023

Ttd

Fajar Maulana Herawan - 13521080