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

Fajar Maulana Herawan - 13521080
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jalan Ganesha 10 Bandung
E-mail (gmail): 13521080@std.stei.itb.ac.id

***Abstract*—*Tower of Hanoi* merupakan salah satu topik yang seringkali menjadi pokok pembahasan di dunia Informatika. *Tower of Hanoi* merupakan salah satu contoh permainan klasik tradisional yang menerapkan algoritma rekursif. Makalah ini akan membahas metode otomatisasi solusi *Tower of Hanoi* menggunakan 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. Oleh karena itu, pada makalah ini, akan dibahas pendekatan BFS dengan menggunakan suatu proses *enqueue* dan *dequeue* pada *queue* untuk menerapkan algoritma rekursif dengan *Breadth-First Search* (BFS).**

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 adalah 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. 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.

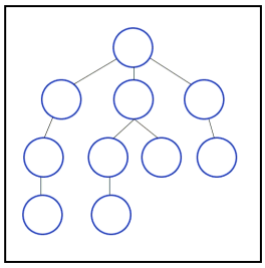
Permainan kombinatorial merupakan salah satu cabang ilmu kombinatorika yang mempelajari permainan dengan *perfect information* atau dengan kata lain seluruh pemain memiliki informasi lengkap mengenai kondisi dan keadaan permainan. Selain itu, permainan kombinatorial juga bersifat deterministik, yang 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*.

Adapun penyelesaian permainan kombinatorial ini dapat menggunakan algoritma *Breadth-First Search* (BFS). Algoritma ini dianggap efektif untuk menyelesaikan permasalahan *Tower of Hanoi*. Selain itu, algoritma BFS ini dipilih karena mudah dimengerti dan memiliki solusi yang cukup optimal. Algoritma lain yang dapat digunakan selain BFS ada DFS atau A*.Makalah ini akan lebih fokus ke penyelesaian dengan algoritma BFS.

II. Landasan Teori

A. *Graf Traversal*

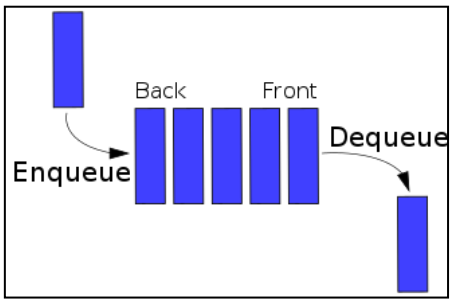
Graf Traversal adalah sebuah konsep dalam ilmu komputer untuk mengunjungi setiap simpul dari suatu graf tepat satu kali dalam urutan yang jelas atau sistematis dari sebuah grafik. Pada graf traversal, akan dilakukan pula proses menghitung urutan simpul dengan menggunakan teknik *traverse*. Proses dimulai dengan mengunjungi simpul awal dan berakhir pada simpul akhir yang telah ditentukan sebelumnya.Setiap simpul yang telah dikunjungi akan disimpan informasi-nya agar tidak terjadi sebuah kondisi *infinite loop* akibat dari suatu simpul dikunjungi berulang-ulang kali. Proses pencatatan simpul yang sudah dikunjungi dapat menggunakan *boolean* , dengan true pada simpul yang telah dikunjungi dan false pada simpul 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:

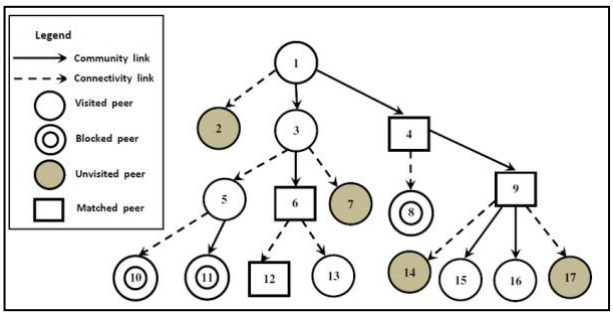
1. *Simple Queue* : Variasi *simple queue* atau yang biasa disebut sebagai *linear queue* merupakan variasi paling dasar dari suatu struktur data queue. Pada variasi ini, penambahan elemen akan dilakukan dengan operasi `*enqueue*` terhadap elemen terakhir, sementara penghapusan elemen akan dilakukan 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 *traversal* pada graf (pencarian solusi). Dalam algoritma BFS, graf *G = (V, E)* dapat direpresentasikan dengan matriks ketetanggaan.



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

Algoritma ini akan memulai penelusuran dari simpul v yang merupakan simpul akar.Kemudian algoritma akan mengunjungi semua simpul yang bertetanggan dengan simpul v terlebih dahulu. Proses akan terus berlangsung dengan mengunjungi simpul simpul yang belum dikunjungi dari simpul yang bertetangaa dan telah dikunjungi hingga ditemukan simpul goal atau tidak ada lagi simpul yang belum dikunjungi.

Selain itu, kita juga perlu menyimpan simpul mana saja yang sudah dikunjungi dengan menggunakan *hashmap* atau *array of boolean*. Berikut adalah pseudocode untuk prosedur *breadth-first-search* (BFS).

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

KAMUS
  { Variabel }
  q : queue
  w : simpul
  dikunjungi : hashTable

  procedure BuatAntrian(input/output q: queue)
  {membuat antrian kosong, kepala(q) diisi 0}
 procedure MasukAntrian(input/output q: queue, input v: simpul)
  { Memasukkan v ke dalam q dengan aturan FIFO (pada posisi belakang)}
  function HapusAntrian(q: queue) → simpul
  { Menghapus simpul dari kepala q dengan aturan FIFO
    dan mengembalikan simpul yang dihapus}
  function AntrianKosong(q: queue) → boolean
  { Mengembalikan True q kosong,false jika sebaliknya }
  procedure createHashTable(input/output T: hashTable)
   { Inisialisasi hashTable dengan memasukan nilai false sejumlah simpul di dalam graf}
  procedure setHashTable(input/output T: hashTable, input v: simpul, input stat: boolean )
  { Mengubah value dari element T pada key v
    I.S. T sudah terdefinisi
    F.S. Elemen T pada key v diubah menjadi stat }
  function getHashTable(T: hashTable, input v: simpul) → boolean
  { Mengembalikan elemen T pada key v }

ALGORITMA
  createHashTable(dikunjungi) { Inisialisasi dikunjungi }

  BuatAntrian(q) { buat antrian kosong }

  write(v) { Mengunjungi simpul pertama }


  MasukAntrian(q, v)
  setHashTable(dikunjungi, v, True)

   { Kunjungi semua simpul graf selama antrian belum kosong}
  while (not AntrianKosong(q)) do
      v ← HapusAntrian(q)
      for setiap simpul w yang bertetangga dengan simpul v do
          if (not getHashTable(dikunjungi, w)) then
              write(w)
              MasukAntrian(q, w)
              setHashTable(dikunjungi, w, True)

  { AntrianKosong(q)}
```

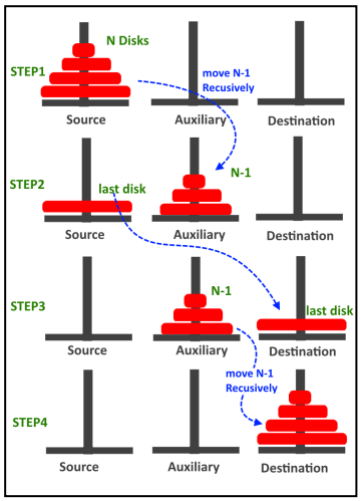
Dengan menabahkan kode pada algoritma *breadth-first-search* (BFS), penulis mendapatkan simpul goal. Implementasi dari kode tersebut dapat dilihat di bawah

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

KAMUS
  { Variabel }
  q : queue
  w : simpul
  found : boolean
  dikunjungi : hashTable

  procedure BuatAntrian(input/output q: queue)
  {membuat antrian kosong, kepala(q) diisi 0}
 procedure MasukAntrian(input/output q: queue, input v: simpul)
  { Memasukkan v ke dalam q dengan aturan FIFO (pada posisi belakang)}
  function HapusAntrian(q: queue) → simpul
  { Menghapus simpul dari kepala q dengan aturan FIFO
    dan mengembalikan simpul yang dihapus}
  function AntrianKosong(q: queue) → boolean
  { Mengembalikan True q kosong,false jika sebaliknya }
  procedure createHashTable(input/output T: hashTable)
   { Inisialisasi hashTable dengan memasukan nilai false sejumlah simpul di dalam graf}
  procedure setHashTable(input/output T: hashTable, input v: simpul, input stat: boolean )
  { Mengubah value dari element T pada key v
    I.S. T sudah terdefinisi
    F.S. Elemen T pada key v diubah menjadi stat }
  function getHashTable(T: hashTable, input v: simpul) → boolean
  { Mengembalikan elemen T pada key v }

ALGORITMA
  createHashTable(dikunjungi) { Inisialisasi dikunjungi }

  BuatAntrian(q) { buat antrian kosong }

  write(v) { Mengunjungi simpul pertama }
  found ← false
  MasukAntrian(q, v)
  setHashTable(dikunjungi, v, True)

   { Kunjungi semua simpul graf selama antrian belum kosong}
  while (not AntrianKosong(q)) do
      v ← HapusAntrian(q)
     if(v = goal) then
        found ← true
     if(not(found)) then
        for setiap simpul w yang bertetangga dengan simpul v do
          if (not getHashTable(dikunjungi, w)) then
              write(w)
              MasukAntrian(q, w)
              setHashTable(dikunjungi, w, True)

  { AntrianKosong(q)}
```

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

Contoh penerapan algoritma *breadth-first-search* (BFS) dengan solusi rekursif dalam *source code file* pada bahasa *python*:

```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 Algoritma BFS dalam Pembuatan Permainan *Puzzle Tower of Hanoi*

A. *Struktur Direktori Program*

Struktur dari *directory program* yang diimplementasikan dibagi menjadi tiga folder yakni `doc`, `img`, dan `src`. Selain itu, terdapat dua file yaitu `README.md` sebagai petunjuk menjalankan permainan dan `requirements.txt sebagai beberapa *library python* yang digunakan dalam permainan ini.Folder doc berisi makalah untuk memenuhi tugas mata kuliah Strategi Algoritma IF2211 Tahun 2022/2023. Folder img berisi gambar,favicon, dan gif yang digunakan baik di dalam file `README.md` maupun di dalam implementasi program.Terakhit ada folder `*src*` yang berisi 2 program utama dengan bahasa *python*, bernama `*main.py*` dan `*disk.py*`. Kedua file tersebut untuk menjalankan Permainan *Tower of Hanoi*. Program menggunakan *library PyGame* sebagai GUI untuk permainan dan algoritma *Breadth-First-Search* (BFS) untuk mencari solusi dari permainan tersebut.

B. *Implementasi Fungsi Utama*

Fungsi utama untuk mencari solusi dalam kode program ini terbagi menjadi dua yakni `*preBFS*` dan `*BFS*` yang berada pada file `main.py` di folder `src`. Fungsi `*preBFS(p_s, p_f, pools)*`merupakan fungsi yang digunakan untuk mengeksekusi pemindahan kontrol sebelum menjalankan algoritma BFS. Fungsi ini memiliki tiga parameter dengan masing masing 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 dari 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*`. Kode fungsi `preBFS` dapat dilihat pada gambar 3.1

```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

Selain itu, fungsi `*BFS(pool_stat)*`merupakan implementasi algoritma BFS untuk mencari solusi optimal dalam permainan *Tower of Hanoi*. Fungsi `BFS` memiliki 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*. Kode `BFS` dapat dilihat pada gambar 3.2

```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 Lain-Lain*

Adapun fungsi lain yang digunakan untuk program ini yakni *gameInit()*, *addControl(pool_ind)*, dan *listPool()*.

```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*. Implementasi dari fungsi ini dapat dilihat pada gambar 3.3 .

```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

Kemudian, ada fungsi `*addControl(pool_ind)*` yang digunakan untuk menambahkan kontrol (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. Implementasi dari fungsi ini dapat dilihat pada gambar 3.4 .

Dan terakhir ada fungsi `*listPoll()*` yang berguna untuk menghasilkan daftar pool saat ini. Implementasi dari fungsi ini dapat dilihat pada gambar 3.5

```python
# 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. *Implementasi 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. Implementasi kelas disk dapat dilihat pada gambar 3.6 .

```python
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. Daftar Library

Di dalam proses implementasi dibutuhkan beberapa library pendukung untuk membuat permainan baik tampilan maupun algoritma.Adapun library *python* yang digunakan dalam pembuatan permainan,yakni

1. Pygame
2. Numpy
3. Copy
4. Time
5. OS
6. SYS

F. Fitur – Fitur Program

Program memiliki empat fitur ,antara lain

1. Fitur untuk memindahkah piringan secara manual,
   fitur ini dapat dilakukan dengan menekan `*left click*` pada *mouse* dan menekannya kembali pada pool tujuan

2. Fitur untuk mencari solusi otomatis dengan algoritma BFS,
   fitur ini dapat dilakukan dengan menekan tombol `*s*` pada *keyboard*

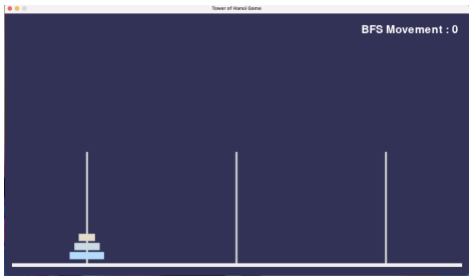
3. Fitur untuk menambah dan mengurangi jumlah piringan,
   fitur ini dapat dilakukan dengan menekan tombol `*left/right arrow*` pada *keyboard*

4. Fitur untuk mempercepat dan memperlambat *game speed*,
   fitur ini dapat dilakukan dengan menekan tombol `*up/down arrow*` pada *keyboard*

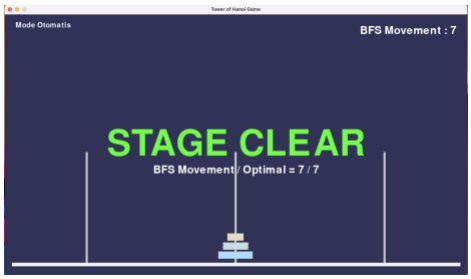
IV. HASIL PENGUJIAN PROGRAM

A. Uji Coba Program

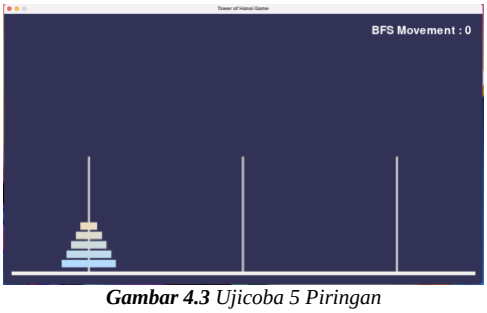
Proses pengujian akan dilakukan sebanyak 3 kali berdasarkan jumlah piringan yang digunakan dalam permainan. Berikut adalah hasil percobaan yang dilakukan:



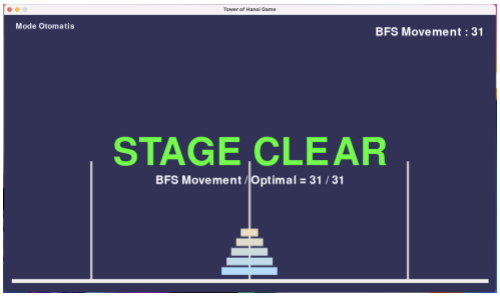
Gambar 4.1 *Ujicoba 3 Piringan*
Sumber : Dokumen Pribadi



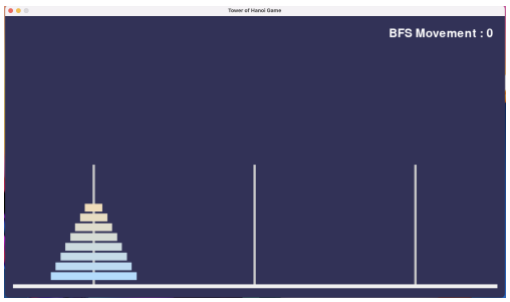
Gambar 4.2 *Hasil Ujicoba 3 Piringan*
Sumber : Dokumen Pribadi



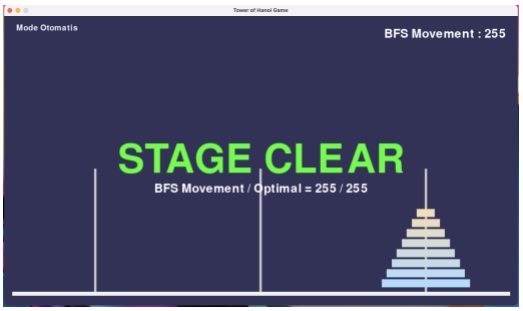
Gambar 4.3 *Ujicoba 5 Piringan*
Sumber : Dokumen Pribadi



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

Hasil dari tiga percobaan yang telah dilakukan terkait jumlah langkah pada setiap variasi, 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. Hasil dari percobaan dapat dilihat pada tabel 4.1.

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 adalah 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.Berdasarkan hasil uji, algoritma BFS untuk menyelesaikan permasalahan *tower of hanoi* menghasilkan solusi yang optimal yakni seluruhnya hasil uji coba berjumlah sama dengan hasil perhitungan menggunakan persamaan 2^N *- 1*.Dengan demikian, Algoritma BFS untuk mencari solusi otomatis dari permainan *Tower of Hanoi* dapat diimplementasikan.

UCAPAN TERIMA KASIH

Puji syukur kepada Tuhan yang Maha Esa karena-Nya penulis dapat menyelesaikan makalah dengan judul "Pembuatan Permainan serta Solusi Otomatis pada *Puzzle Tower of Hanoi* dengan Algoritma BFS".Penulis juga ini berterima kasih kepada Dr. Nur Ulfa Maulidevi, S.T., M.Sc. selaku dosen pengampu mata kuliah IF2211 Strategi Algoritma. Penulis juga mengucapkan terima kasih kepada Dr. Ir. Rinaldi Munir, M.T sebagai salah satu sumber referensi dari makalah ini. Selain itu, penulis juga mengucapkan terima kasih kepada pihak-pihak luar yang bersangkutan. Akhir kata, penulis ingin mengucapkan permohonan maaf jika ada kesalahan-kesalahan dalam penulisan makalah ini.

LAMPIRAN

Berikut adalah tautan untuk *source code* dari program yang penulis buat

https://github.com/fajarmhrwn/TowerofHanoiSolutionBFS

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 18 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 18 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 18 Mei 2023.

[4] The Shand Book. 2020. *Breadth First Search*. https://www.thedshandbook.com/breadth-first-search/, diakses pada 18 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 18 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, 21 Mei 2023

Fajar Maulana Herawan - 13521080