Perbandingan Kecepatan Algoritma BFS dan DFS dalam penyelesaian solusi *Water Jug Problem*

Dendy Suprihady/ 13514070
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
dendyliu666@yahoo.com

***Abstract*—Dengan berkembangnya dunia teknologi informasi banyak permasalahan seperti TSP (Travelling Salesman Problem), KnapSack, 15-*puzzle* dan sebagainya yang sudah dipecahkan dengan memanfaatkan teknologi informasi. Kali ini, penulis membahas mengenai WaterJug Problem dimana penulis akan mencoba membandingkan algoritma BFS dan DFS dalam penyelesaian masalah WaterJug Problem dengan implementasi yang dilakukan pada pemrograman bahasa Java.**

Kata kunci: DFS, BFS, Water Jug Problem.

I. PENDAHULUAN

Sejak dulu WaterJug Problem sudah sangat dikenal dikalangan pecinta riddle problem dan masalah ini sering dijadikan sebagai soal dalam kuis atau teka teki untuk mengasah logika berpikir.Dengan ilmu teknologi informasi yang sudah berkembang pesat masalah ini masih sering dijadikan bahan pembelajaran dalam dunia informatika terutama pada bidang *artificial intelegent*. Dalam proses penyelesaian masalah ini sering sekali digunakan algoritma DFS(Depth First-Search) dan BFS (Breadth First-Search). DFS dan BFS adalah merupakan algoritma penulusuran graph yang sangat populer di dunia informatika, algoritma ini banyak dipakai untuk memecahkan berbagai permasalahan yang dapat diimplementasikan kedalam sebuah graph dengan membagi masalah kedalam beberapa state yang memungkinkan hingga ditemukannya state finish atau state solusi. Dengan melihat hal tersebut penulis ingin mencoba membandingkan kecepatan kedua algoritma tersebut dalam memecahkan masalah WaterJug Problem yang diimplementasikan dalam program Java.

II. DASAR TEORI

1.Algoritma BFS

1.1 Pengertian

Breadth-first search adalah algoritma yang melakukan pencarian secara melebar yang mengunjungi simpul secara *preorder* yaitu mengunjungi suatu simpul kemudian mengunjungi semua simpul yang bertetangga dengan simpul tersebut terlebih dahulu. Selanjutnya, simpul yang belum dikunjungi dan bertetangga dengan simpulsimpul yang tadi dikunjungi , demikian seterusnya. Jika graf berbentuk pohon berakar, maka semua simpul pada aras *d* dikunjungi lebih dahulu sebelum simpul-simpul pad aras *d*+1.

Algoritma ini memerlukan sebuah antrian *q* untuk menyimpan simpul yang telah dikunjungi. Simpulsimpul ini diperlukan sebagai acuan untuk mengunjungi simpul-simpul yang bertetanggaan dengannya. Tiap simpul yang telah dikunjungu masuk ke dalam antrian hanya satu kali. Algoritma ini juga membutuhkan table Boolean untuk menyimpan simpul yang te lah dikunjungi sehingga tidak ada simpul yang dikunjungi lebih dari satu kali.

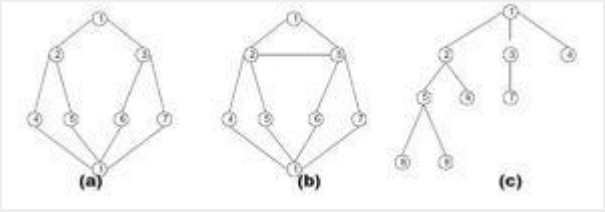
1.2 Cara Kerja BFS

Dalam algoritma BFS, simpul anak yang telah dikunjungi disimpan dalam suatu antrian. Antrian ini digunakan untuk mengacu simpul-simpul yang bertetangga dengannya yang akan dikunjungi kemudian sesuai urutan pengantrian.

Untuk memperjelas cara kerja algoritma BFS beserta antrian yang digunakannya, berikut langkah-langkah algoritma BFS:

1. Masukkan simpul ujung (akar) ke dalam antrian
2. Ambil simpul dari awal antrian, lalu cek apakah simpul merupakan solusi
3. Jika simpul merupakan solusi, pencarian selesai dan hasil dikembalikan.
4. Jika simpul bukan solusi, masukkan seluruh simpul yang bertetangga dengan simpul tersebut (simpul anak) ke dalam antrian
5. Jika antrian kosong dan setiap simpul sudah dicek, pencarian selesai dan mengembalikan hasil solusi tidak ditemukan
6. Ulangi pencarian dari langkah kedua

Contohnya terlihat dibawah ini:



Maka penyelesaiannya adalah:

Gambar (a) BFS(1): 1, 2, 3, 4, 5, 6, 7, 1.

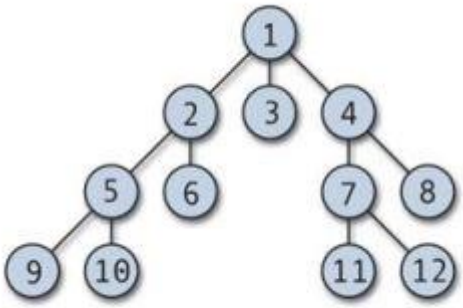
Gambar (b) BFS(1): 1, 2, 3, 4, 5, 6, 7, 1

Gambar (c) BFS(1): 1, 2, 3, 4, 5, 6, 7, 8, 9

1.3 Pencarian Lintasan Terpendek dengan BFS

Adapun contoh untuk mencari lintasan terpendek dengan menggukan algoritma BFS adalah sebagai berkut:

Diketahui sebuah kota, dengan memiliki inisial seperti yang ditunjukkan dibawah ini. Jarak antar kota dibentuk dengan sebuah graph terlihat dibawah:



Pertanyaan: sebutkan rute yang akan ditempuh untuk mencapai kota no. 8. Titik awal perjalanan adalah kota no. 1. Gunakan algoritma BFS!

Maka dengan menggunakan algoritma BFS, rute tercepat yang didapat adalah sebagai berikut:

1 – 2 – 3 – 4 – 5 – 6 – 7 – 8

Rute tersebut didapat dari pencarian secara melebar. Hal; tersebut dapat dijabarkan sebagai berikut:

–Pertama-tama, pointer menunujuk pada daun yang ada sebelah kanan, yaitu no.2 (1 – 2)

– Setelah itu, proses dilanjutkan pada tetangga no.2 yaitu no.3 (1-2-3) dan selanjutnya mengarah pada tetangga terdekat, yakni no.4 (1-2-3-4).

– Pointer mencari teteangga no.4, namun karna tidak ada, maka pointer kembali ke kota no.2 dan masuk ke daun berikutnya, yakni no.5.

– Proses diulang hingga pointer menunjuk angka 8

1.4 Penerapan BFS dalam Bahasa Algoritma

```
procedure BFS(input v:integer)
{ Traversal graf dengan algoritma pencarian BFS.

  Masukan: v adalah simpul awal kunjungan
  Keluaran: semua simpul yang dikunjungi dicetak ke layar
}
Deklarasi
   w : integer
   q : antrian;

   procedure BuatAntrian(input/output q : antrian)
   { membuat antrian kosong, kepala(q) diisi 0 }

   procedure MasukAntrian(input/output q:antrian, input v:integer)
   { memasukkan v ke dalam antrian q pada posisi belakang }

   procedure HapusAntrian(input/output q:antrian,output v:integer)
   { menghapus v dari kepala antrian q }

   function AntrianKosong(input q:antrian) → boolean
   { true jika antrian q kosong, false jika sebaliknya }

Algoritma:
   BuatAntrian(q)          { buat antrian kosong }

   write(v)                { cetak simpul awal yang dikunjungi }
   dikunjungi[v]←true      { simpul v telah dikunjungi, tandai dengan
                             true}
   MasukAntrian(q,v)       { masukkan simpul awal kunjungan ke dalam
                             antrian}

  { kunjungi semua simpul graf selama antrian belum kosong }
   while not AntrianKosong(q) do
       HapusAntrian(q,v)   { simpul v telah dikunjungi, hapus dari
                             antrian }
       for tiap simpul w yang bertetangga dengan simpul v do
             if not dikunjungi[w] then
                write(w)     {cetak simpul yang dikunjungi}
                MasukAntrian(q,w)
                dikunjungi[w]←true
             endif
       endfor
   endwhile
  { AntrianKosong(q) }
```

2.Algoritma DFS

2.1 Pengertian

Algoritma Depth First Search (DFS) adalah salah satu algoritma pencarian solusi yang digunakan di dalam kecerdasan buatan. Algoritma ini termasuk salah satu jenis uninformed algorithm yaitu algoritma yang melakukan pencarian dalam urutan tertentu tetapi tidak memiliki informasi apa-apa sebagai dasar pencarian kecuali hanya mengikuti pola yang diberikan.

Di dalam DFS, pencarian dilakukan pada suatu struktur pohon yaitu kumpulan semua kondisi yang mungkin yang diimplementasikan dalam sebuah struktur pohon. Paling atas adalah akar (root) yang berisi kondisi awal pencarian (initial state) dan di bawahnya adalah kondisi-kondisi berikutnya sampai kepada kondisi tujuan (goal state).

2.2 Cara Kerja DFS

Untuk melakukan pencarian,DFS menggunakan cara sebagaiberikut:

1. Masukkan Initial State pada Tumpukan.
2. Periksa apakah ada data di tumpukan.
3. Jika tidak, maka solusi tidak ditemukan, dan proses berhenti.

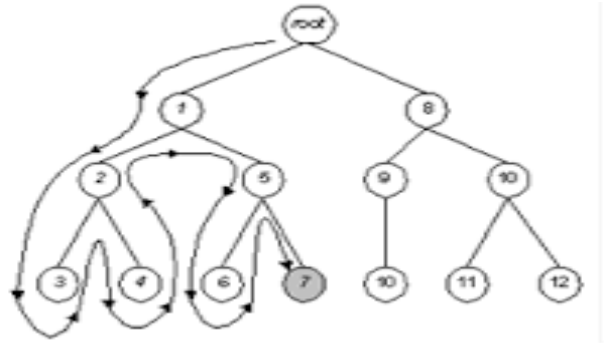
4. Jika ya, Ambil state pada tumpukan paling atas.
5. Bandingkan State tersebut apakah sama dengan Goal State
6. Jika sama, maka solusi ditemukan dan proses berakhir.
7.Jika tidak, ekspansikan state tersebut.
6. Masukkan seluruh state hasil ekspansi ke dalam tumpukan.
7.Kembali ke langkah 2.

Pada prinsipnya, DFS ini menggunakan tumpukan untuk menyimpan seluruh state yang ditemukan atau bisa dikatakan bahwa DFS menggunakan metode LIFO (*Last In First Out*).

2.3 Pencarian Lintasan Terpendek dengan DFS

Adapun contoh untuk mencari lintasan terpendek dengan menggukan algoritma DFS adalah sebagai berkut:

Diketahui sebuah kota, dengan memiliki inisial seperti yang ditunjukkan dibawah ini. Jarak antar kota dibentuk dengan sebuah graph terlihat dibawah:



Pertanyaan: sebutkan rute yang akan ditempuh untuk mencapai kota no. 7. Titik awal perjalanan adalah kota root 1. Gunakan algoritma DFS!

Maka dengan menggunakan algoritma BFS, rute tercepat yang didapat adalah sebagai berikut:

Root -1 - 2 - 3 - 4 - 5 - 6 - 7

Rute tersebut didapat dari pencarian secara mendalam. Hal; tersebut dapat dijabarkan sebagai berikut:

–Pertama-tama, pointer menunujuk pada daun yang ada sebelah kanan, yaitu no.1 (Root -1)

– Setelah itu, proses dilanjutkan pada anak no.1 yaitu no.2 (Root-1-2) dan selanjutnya mengarah pada anak dari simpul no.2, yakni no.3 (Root-1-2-3).

– Pointer mencari anak no.3, namun karna tidak ada, maka pointer kembali ke kota no.2 dan masuk ke anak yang berikutnya, yakni no.4 (Root-1-2-3-4).

– Proses diulang hingga pointer menunjuk angka 7

2.3 Penerapan DFS dalam Bahasa Algoritma

```
procedure DFS(input v:integer)
{Mengunjungi seluruh simpul graf dengan algoritma pencarian DFS

Masukan: v adalah simpul awal kunjungan
Keluaran: semua simpulyang dikunjungi ditulis ke layar
}
Deklarasi
   w : integer

Algoritma:
   write(v)
   dikunjungi[v]←true
   for w←1 to n do
      if A[v,w]=1 then {simpul v dan simpul w bertetangga }
         if not dikunjungi[w] then
            DFS(w)
         endif
      endif
   endfor
```

3.Water Jug Problem

Permasalahan teko air merupakan suatu permasalahan klasik dalam bidang ilmu Artificial Intelligence (AI). Permasalahan ini dapat diilustrasikan seperti berikut, terdapat 2 buah teko air masing-masing memiliki kapasitas x dan y liter. Permasalahannya adalah bagaimana mendapatkan air sebanyak n liter dengan menggunakan bantuan kedua teko air tersebut dan mengambil asumsi bahwa sumber air tidak terbatas. Aksi-aksi yang dapat dilakukan, antara lain mengisi teko air, mengosongkan teko air dan menuangkan isi teko air ke teko air lain.

Permasalahan ini dapat diselesaikan dengan menerapkan konsep AI yaitu dengan bantuan pohon pelacakan dan menerapkan metode pencarian melebar pertama (breadth-first search / BFS). Pencarian solusi dimulai dari kondisi dimana kedua teko kosong (node akar dari pohon pelacakan). Proses dilanjutkan dengan menggambarkan kondisi (state) berikutnya (dengan melakukan aksi terhadap state sebelumnya) hingga semua state diperiksa dan mendapatkan tujuan (goal state).

III. PENYELESAIAN MASALAH

Untuk melakukan penyelesaian masalah yang telah dijabarkan di atas maka permasalahan water jug problem akan diimplementasikan ke dalam sebuah program dalam bahasa C++.

- Struktur Data state

```
// Representation of a state (x, y)
// x and y are the amounts of water in litres in the two jugs respectively
struct state {
    int x, y;

    // Used by map to efficiently implement lookup of seen states
    bool operator < (const state& that) const {
        if (x != that.x) return x < that.x;
        return y < that.y;
    }
};
```

- Source Code prosedur DFS

```
void dfs(state start, stack <pair <state, int> >& path)    {
    stack <state> s;
    state goal = (state) {-1, -1};

    // Stores seen states so that they are not revisited and
    // maintains their parent states for finding a path through
    // the state space
    // Mapping from a state to its parent state and rule no. that
    // led to this state
    map <state, pair <state, int> > parentOf;

    s.push(start);
    parentOf[start] = make_pair(start, 0);

    while (!s.empty())    {
        // Get the state at the front of the stack
        state top = s.top();
        s.pop();

        // If the target state has been found, break
        if (top.x == target || top.y == target) {
            goal = top;
            break;
        }

        // Find the successors of this state
        // This step uses production rules to produce successors of the
current state
        // while pruning away branches which have been seen before

        // Rule 1: (x, y) -> (capacity_x, y) if x < capacity_x
        // Fill the first jug
        if (top.x < capacity_x)  {
            state child = (state) {capacity_x, top.y};
            // Consider this state for visiting only if it has not been
visited before
            if (parentOf.find(child) == parentOf.end()) {
                s.push(child);
                parentOf[child] = make_pair(top, 1);
            }
        }

        // Rule 2: (x, y) -> (x, capacity_y) if y < capacity_y
        // Fill the second jug
        if (top.y < capacity_y)  {
            state child = (state) {top.x, capacity_y};
            if (parentOf.find(child) == parentOf.end()) {
                s.push(child);
                parentOf[child] = make_pair(top, 2);
            }
        }

        // Rule 3: (x, y) -> (0, y) if x > 0
        // Empty the first jug
        if (top.x > 0)  {
            state child = (state) {0, top.y};
            if (parentOf.find(child) == parentOf.end()) {
                s.push(child);
                parentOf[child] = make_pair(top, 3);
            }
        }

        // Rule 4: (x, y) -> (x, 0) if y > 0
        // Empty the second jug
        if (top.y > 0)  {
            state child = (state) {top.x, 0};
            if (parentOf.find(child) == parentOf.end()) {
                s.push(child);
                parentOf[child] = make_pair(top, 4);
            }
        }

        // Rule 5: (x, y) -> (min(x + y, capacity_x), max(0, x + y -
capacity_x)) if y > 0
        // Pour water from the second jug into the first jug until the first
jug is full
        // or the second jug is empty
        if (top.y > 0)  {
            state child = (state) {min(top.x + top.y, capacity_x), max(0,
top.x + top.y - capacity_x)};
            if (parentOf.find(child) == parentOf.end()) {
                s.push(child);
                parentOf[child] = make_pair(top, 5);
            }
        }

        // Rule 6: (x, y) -> (max(0, x + y - capacity_y), min(x + y,
capacity_y)) if x > 0
        // Pour water from the first jug into the second jug until the second
jug is full
        // or the first jug is empty
        if (top.x > 0)  {
            state child = (state) {max(0, top.x + top.y - capacity_y),
min(top.x + top.y, capacity_y)};
            if (parentOf.find(child) == parentOf.end()) {
                s.push(child);
                parentOf[child] = make_pair(top, 6);
            }
        }
    }

    // Target state was not found
    if (goal.x == -1 || goal.y == -1)
        return;

    // backtrack to generate the path through the state space
    path.push(make_pair(goal, 0));
    // remember parentOf[start] = (start, 0)
    while (parentOf[path.top().first].second != 0)
        path.push(parentOf[path.top().first]);
}
```

- Source Code prosedur BFS

```
void bfs(state start, stack <pair <state, int> >& path)   {
    queue <state> q;
    state goal = (state) {-1, -1};

    // Stores seen states so that they are not revisited and
    // maintains their parent states for finding a path through
    // the state space
    // Mapping from a state to its parent state and rule no. that
    // led to this state
    map <state, pair <state, int> > parentOf;

    q.push(start);
    parentOf[start] = make_pair(start, 0);

    while (!q.empty())    {
        // Get the state at the front of the queue
        state top = q.front();
        q.pop();

        // If the target state has been found, break
        if (top.x == target || top.y == target) {
            goal = top;
            break;
        }
        // Find the successors of this state
        // This step uses production rules to prune the search space

        // Rule 1: (x, y) -> (capacity_x, y) if x < capacity_x
        // Fill the first jug
        if (top.x < capacity_x)  {
            state child = (state) {capacity_x, top.y};
            // Consider this state for visiting only if it has not been
 visited before
            if (parentOf.find(child) == parentOf.end()) {
                q.push(child);
                parentOf[child] = make_pair(top, 1);
            }
        }

        // Rule 2: (x, y) -> (x, capacity_y) if y < capacity_y
        // Fill the second jug
        if (top.y < capacity_y)  {
            state child = (state) {top.x, capacity_y};
            if (parentOf.find(child) == parentOf.end()) {
                q.push(child);
                parentOf[child] = make_pair(top, 2);
            }
        }

// Rule 3: (x, y) -> (0, y) if x > 0
// Empty the first jug
if (top.x > 0)  {
    state child = (state) {0, top.y};
    if (parentOf.find(child) == parentOf.end()) {
        q.push(child);
        parentOf[child] = make_pair(top, 3);
    }
}

// Rule 4: (x, y) -> (x, 0) if y > 0
// Empty the second jug
if (top.y > 0)  {
    state child = (state) {top.x, 0};
    if (parentOf.find(child) == parentOf.end()) {
        q.push(child);
        parentOf[child] = make_pair(top, 4);
    }
}

        // Rule 5: (x, y) -> (min(x + y, capacity_x), max(0, x + y -
capacity_x)) if y > 0
        // Pour water from the second jug into the first jug until the first
jug is full
        // or the second jug is empty
        if (top.y > 0)  {
            state child = (state) {min(top.x + top.y, capacity_x), max(0,
top.x + top.y - capacity_x)};
            if (parentOf.find(child) == parentOf.end()) {
                q.push(child);
                parentOf[child] = make_pair(top, 5);
            }
        }

        // Rule 6: (x, y) -> (max(0, x + y - capacity_y), min(x + y,
capacity_y)) if x > 0
        // Pour water from the first jug into the second jug until the second
jug is full
        // or the first jug is empty
        if (top.x > 0)  {
            state child = (state) {max(0, top.x + top.y - capacity_y),
min(top.x + top.y, capacity_y)};
            if (parentOf.find(child) == parentOf.end()) {
                q.push(child);
                parentOf[child] = make_pair(top, 6);
            }
        }
    }


    // Target state was not found
    if (goal.x == -1 || goal.y == -1)
        return;

    // backtrack to generate the path through the state space
    path.push(make_pair(goal, 0));
    // remember parentOf[start] = (start, 0)
    while (parentOf[path.top().first].second != 0)
        path.push(parentOf[path.top().first]);
}
```

- Main Program

```
int main()  {
    stack <pair <state, int> > path;

    printf("Enter the capacities of the two jugs : ");
    scanf("%d %d", &capacity_x, &capacity_y);
    printf("Enter the target amount : ");
    scanf("%d", &target);

    //bfs((state) {0, 0}, path); FOR BFS ALGORITHM
    //dfs((state) {0, 0}, path); FOR DFS ALGORITHM
    if (path.empty())
        printf("\nTarget cannot be reached.\n");
    else    {
        printf("\nNumber of moves to reach the target : %d\nOne path to the
 target is as follows :\n", path.size() - 1);
        while (!path.empty())   {
            state top = path.top().first;
            int rule = path.top().second;
            path.pop();

            switch (rule)   {
                case 0: printf("State : (%d, %d)\n#\n", top.x, top.y);
                        break;
                case 1: printf("State : (%d, %d)\nAction : Fill the first
jug\n", top.x, top.y);
                        break;
                case 2: printf("State : (%d, %d)\nAction : Fill the second
jug\n", top.x, top.y);
                        break;
                case 3: printf("State : (%d, %d)\nAction : Empty the first
jug\n", top.x, top.y);
                        break;
                case 4: printf("State : (%d, %d)\nAction : Empty the second
jug\n", top.x, top.y);
                        break;
                case 5: printf("State : (%d, %d)\nAction : Pour from second
jug into first jug\n", top.x, top.y);
                        break;
                case 6: printf("State : (%d, %d)\nAction : Pour from first
jug into second jug\n", top.x, top.y);
                        break;
            }
        }
    }

    return 0;
}
```

- Uji Coba

  Pada bagian uji coba kita akan menggunakan beberapa kasus dengan mengubah – ngubah nilai kapasitas dari 2 water jug dan nilai target.

  -Untuk kapasitas Jug x= 1,Jug y= 3,dan target= 2:

  Hasil run BFS

```
C:\Users\Asus X550ZE\Desktop\Water Jug Problem>waterjugbfs
Enter the capacities of the two jugs : 1 3
Enter the target amount : 2

Number of moves to reach the target : 2
One path to the target is as follows :
State : (0, 0)
Action : Fill the second jug
State : (0, 3)
Action : Pour from second jug into first jug
State : (1, 2)
#
```

  Hasil run DFS

```
C:\Users\Asus X550ZE\Desktop\Water Jug Problem>waterjugdfs
Enter the capacities of the two jugs : 1 3
Enter the target amount : 2

Number of moves to reach the target : 2
One path to the target is as follows :
State : (0, 0)
Action : Fill the second jug
State : (0, 3)
Action : Pour from second jug into first jug
State : (1, 2)
#
```

  -Untuk kapasitas Jug x= 3,Jug y= 5,dan target= 4 :

  Hasil run BFS

```
C:\Users\Asus X550ZE\Desktop\Water Jug Problem>waterjugbfs
Enter the capacities of the two jugs : 3 5
Enter the target amount : 4

Number of moves to reach the target : 6
One path to the target is as follows :
State : (0, 0)
Action : Fill the second jug
State : (0, 5)
Action : Pour from second jug into first jug
State : (3, 2)
Action : Empty the first jug
State : (0, 2)
Action : Pour from second jug into first jug
State : (2, 0)
Action : Fill the second jug
State : (2, 5)
Action : Pour from second jug into first jug
State : (3, 4)
#
```

  Hasil run DFS

```
C:\Users\Asus X550ZE\Desktop\Water Jug Problem>waterjugdfs
Enter the capacities of the two jugs : 3 5
Enter the target amount : 4

Number of moves to reach the target : 6
One path to the target is as follows :
State : (0, 0)
Action : Fill the second jug
State : (0, 5)
Action : Pour from second jug into first jug
State : (3, 2)
Action : Empty the first jug
State : (0, 2)
Action : Pour from second jug into first jug
State : (2, 0)
Action : Fill the second jug
State : (2, 5)
Action : Pour from second jug into first jug
State : (3, 4)
#
```

  -Untuk kapasitas Jug x= 5,Jug y= 13,dan target= 12 :

  Hasil run BFS

```
C:\Users\Asus X550ZE\Desktop\Water Jug Problem>waterjugbfs
Enter the capacities of the two jugs : 5 13
Enter the target amount : 12

Number of moves to reach the target : 12
One path to the target is as follows :
State : (0, 0)
Action : Fill the first jug
State : (5, 0)
Action : Pour from first jug into second jug
State : (0, 5)
Action : Fill the first jug
State : (5, 5)
Action : Pour from first jug into second jug
State : (0, 10)
Action : Fill the first jug
State : (5, 10)
Action : Pour from first jug into second jug
State : (2, 13)
Action : Empty the second jug
State : (2, 0)
Action : Pour from first jug into second jug
State : (0, 2)
Action : Fill the first jug
State : (5, 2)
Action : Pour from first jug into second jug
State : (0, 7)
Action : Fill the first jug
State : (5, 7)
Action : Pour from first jug into second jug
State : (0, 12)
#
```

Hasil run DFS

```
C:\Users\Asus X550ZE\Desktop\Water Jug Problem>waterjugdfs
Enter the capacities of the two jugs : 5 13
Enter the target amount : 12

Number of moves to reach the target : 22
One path to the target is as follows :
State : (0, 0)
Action : Fill the second jug
State : (0, 13)
Action : Pour from second jug into first jug
State : (5, 8)
Action : Empty the first jug
State : (0, 8)
Action : Pour from second jug into first jug
State : (5, 3)
Action : Empty the first jug
State : (0, 3)
Action : Pour from second jug into first jug
State : (3, 0)
Action : Fill the second jug
State : (3, 13)
Action : Pour from second jug into first jug
State : (5, 11)
Action : Empty the first jug
State : (0, 11)
Action : Pour from second jug into first jug
State : (5, 6)
Action : Empty the first jug
State : (0, 6)
Action : Pour from second jug into first jug
State : (5, 1)
Action : Empty the first jug
State : (0, 1)
Action : Pour from second jug into first jug
State : (1, 0)
Action : Fill the second jug
State : (1, 13)
Action : Pour from second jug into first jug
State : (5, 9)
Action : Empty the first jug
State : (0, 9)
Action : Pour from second jug into first jug
State : (5, 4)
Action : Empty the first jug
State : (0, 4)
Action : Pour from second jug into first jug
State : (4, 0)
Action : Fill the second jug
State : (4, 13)
Action : Pour from second jug into first jug
State : (5, 12)
#
```

Hasil run DFS

```
C:\Users\Asus X550ZE\Desktop\Water Jug Problem>waterjugdfs
Enter the capacities of the two jugs : 7 25
Enter the target amount : 24

Number of moves to reach the target : 46
One path to the target is as follows :
State : (0, 0)
Action : Fill the second jug
State : (0, 25)
Action : Pour from second jug into first jug
State : (7, 18)
Action : Empty the first jug
State : (0, 18)
Action : Pour from second jug into first jug
State : (7, 11)
Action : Empty the first jug
State : (0, 11)
Action : Pour from second jug into first jug
State : (7, 4)
Action : Empty the first jug
State : (0, 4)
Action : Pour from second jug into first jug
State : (4, 0)
Action : Fill the second jug
State : (4, 25)
Action : Pour from second jug into first jug
State : (7, 22)
Action : Empty the first jug
State : (0, 22)
Action : Pour from second jug into first jug
State : (7, 15)
Action : Empty the first jug
State : (0, 15)
Action : Pour from second jug into first jug
State : (7, 8)
Action : Empty the first jug
State : (0, 8)
Action : Pour from second jug into first jug
State : (7, 1)
Action : Empty the first jug
State : (0, 1)
Action : Pour from second jug into first jug
State : (1, 0)
Action : Fill the second jug
State : (1, 25)
Action : Pour from second jug into first jug
State : (7, 19)
Action : Empty the first jug
State : (0, 19)
Action : Pour from second jug into first jug
State : (7, 12)
Action : Empty the first jug
State : (0, 12)
Action : Pour from second jug into first jug
State : (7, 5)
Action : Empty the first jug
State : (0, 5)
Action : Pour from second jug into first jug
State : (5, 0)
Action : Fill the second jug
State : (5, 25)
Action : Pour from second jug into first jug
State : (7, 23)
Action : Empty the first jug
State : (0, 23)
Action : Pour from second jug into first jug
State : (7, 16)
Action : Empty the first jug
State : (0, 16)
Action : Pour from second jug into first jug
State : (7, 9)
Action : Empty the first jug
State : (0, 9)
Action : Pour from second jug into first jug
State : (7, 2)
Action : Empty the first jug
State : (0, 2)
Action : Pour from second jug into first jug
State : (2, 0)
Action : Fill the second jug
State : (2, 25)
Action : Pour from second jug into first jug
State : (7, 20)
Action : Empty the first jug
State : (0, 20)
Action : Pour from second jug into first jug
State : (7, 13)
Action : Empty the first jug
State : (0, 13)
Action : Pour from second jug into first jug
State : (7, 6)
Action : Empty the first jug
State : (0, 6)
Action : Pour from second jug into first jug
State : (6, 0)
Action : Fill the second jug
State : (6, 25)
Action : Pour from second jug into first jug
State : (7, 24)
#
```

-Untuk kapasitas Jug x= 7,Jug y= 25,dan target= 24 :

Hasil run BFS

```
C:\Users\Asus X550ZE\Desktop\Water Jug Problem>waterjugbfs
Enter the capacities of the two jugs : 7 25
Enter the target amount : 24

Number of moves to reach the target : 16
One path to the target is as follows :
State : (0, 0)
Action : Fill the first jug
State : (7, 0)
Action : Pour from first jug into second jug
State : (0, 7)
Action : Fill the first jug
State : (7, 7)
Action : Pour from first jug into second jug
State : (0, 14)
Action : Fill the first jug
State : (7, 14)
Action : Pour from first jug into second jug
State : (0, 21)
Action : Fill the first jug
State : (7, 21)
Action : Pour from first jug into second jug
State : (3, 25)
Action : Empty the second jug
State : (3, 0)
Action : Pour from first jug into second jug
State : (0, 3)
Action : Fill the first jug
State : (7, 3)
Action : Pour from first jug into second jug
State : (0, 10)
Action : Fill the first jug
State : (7, 10)
Action : Pour from first jug into second jug
State : (0, 17)
Action : Fill the first jug
State : (7, 17)
Action : Pour from first jug into second jug
State : (0, 24)
#
```

IV. ANALISIS

Dari bab III bagian uji coba dapat kita lihat bahwa dalam penyelesaian masalah water jug untuk kapasitas x=5 , y=13, target =12 dan kapasitas x=7 , y=25, target=24 algoritma DFS lebih memakan banyak langkah yaitu 22 dan 46 sedangkan untuk algoritma BFS hanya memakan 12 dan 16 langkah, dari hal ini dapat diprediksi untuk nilai x dan y yang selisih semakin besar dan target yang juga semakin besar maka akan semakin banyak langkah yang dibutuhkan bagi algoritma DFS dan semakin besar pula selisih langkah yang dibutuhkan algoritma DFS dan BFS.

Hal ini disebabkan karena algoritma DFS yang melakukan pencarian lebih mendalam dimana ketika solusi pada pohon level atas sebenarnya sudah dapat ditemukan lebih dahulu tetapi algoritma DFS tidak mengeceknya terlebih dahulu, untuk hal ini algoritma BFS yang lebih cepat karena pencarian yang melebar ke semau tetangga simpul pohon yang di cek.

V. KESIMPULAN

Kesimpulan dari permasalahan ini adalah bahwa ternyata untuk menyelesaikan permasalahan Water Jug Problem ternyata lebih efisien atau lebih cepat jika anda menggunakan algoritma BFS ketimbang menggunakan algoritma DFS dikarenakan pencariannya yang melebar.Pada berbagai kasus Water Jug Problem juga tidak ditemukan hasil yang menyatakan langkah yang dilakukan algoritma BFS dalam menyelesaikan masalah lebih banyak dari pada langkah algoritma DFS. Jadi bisa disumpulkan bahwa :

Misal

n = Langkah penyelesaian Water Jug Problem oleh BFS

m= Langkah penyelesaian Water Jug Problem oleh DFS

x= Kapasitas kendi pertama

y= Kapasitas kendi kedua

target= Kapasitas kendi yang ingin dicapai

Maka untuk semua nilai kapasitas x ,y , dan target yang akan menghasilkan solusi dalam permasalahan WaterJug Problem nilai n<=m.

REFERENSI

[1] Munir, Rinaldi. () Diktat Kuliah IF2211 Strategi Algoritma.

[2] http://zeromin0.blogspot.co.id/2011/07/analisa-algoritma-depth-first-search.html#axzz483KjwvJR

[3] http://www.koleksiskripsi.com/2012/01/309-penyelesaian-masalah-kendi-air.html

[4] http://cikalinspirasi.blogspot.co.id/2013/05/algoritma-dept-first-search-dfs.html

[5] https://kartikkukreja.wordpress.com/2013/10/11/water-jug-problem/

[6] https://www.quora.com/How-can-we-use-BFS-and-DFS-to-make-a-tree-for-water-jug-problem

[7] http://timothyariekristiyanto2014.blogspot.co.id/2013/07/water-jug-problem.html

PERNYATAAN

Dengan ini saya menyatakan bahwa makalah yang saya tulis ini adalah tulisan saya sendiri, bukan saduran, atau terjemahandari makalah orang lain, dan bukan plagiasi.

Bandung, 8 Mei 2016

Dendy Suprihady/13514070