Kompleksitas Algoritma dalam menentukan Solvabilitas Sliding N-Puzzle

Audry Nyonata, 13515087
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
audrynyonata@gmail.com, 13515087@std.stei.itb.ac.id

***Abstrak*—N-Puzzle Problem adalah salah satu permasalahan matematika yang terkenal umum ditemukan dalam berbagai implementasi teknologi Kecerdasan Buatan. Makalah ini membahas mengenai sifat solvabilitas puzzle, yaitu apakah puzzle yang memiliki konfigurasi angka-angka tertentu bisa mencapai pola target hanya dengan cara digeser (bertukar tempat dengan ruang kosong). Disediakan pula analisis salah satu contoh source code program dalam menentukan solvabilitas suatu puzzle, kemudian makalah berfokus pada kompleksitas algoritma tersebut yang ditentukan dengan notasi Big-O.**

***Kata Kunci*—Big-O, N-Puzzle, solvabilitas.**

I. PENDAHULUAN

Dalam kehidupan sehari-hari, seringkali kita menemukan beragam variasi permainan asah-otak. Mulai dari catur, dam-daman, tetris, *jigsaw*, sudoku, teka-teki silang, dikenali orang dari segala kalangan tanpa memandang status sosial maupun usia. Terlebih lagi di era teknologi seperti sekarang, para game developer berlomba-lomba menciptakan game untuk menjaring pengguna melalui berbagai platform. Game diciptakan dalam berbagai genre seperti simulasi, arkade, dan puzzle. Permainan yang demikian tidak hanya menjadi sarana hiburan dan rekreasi semata, melainkan juga mampu mengundang rasa keingintahuan, serta melatih kecerdasan otak kita baik dalam aspek ketelitian, daya ingat, refleks, dan sebagainya.

Kini, semakin banyak pula permainan yang melibatkan angka-angka dan permasalahan matematik di dalamnya. Contoh permainan yang berkaitan dengan permasalahan matematika adalah N-puzzle problem, yang sudah lama dikenal oleh masyarakat.

N-Puzzle adalah sebuah permainan puzzle berukuran n x n petak, dengan satu petak kosong, sehingga banyaknya petak pada puzzle sebesar N ($N = n$ x n -1, misal untuk 15-puzzle, ukuran petak 4 x 4). N-Puzzle yang beredar bervariasi baik dari ukuran maupun kondisi kemenangannya. Tetapi cara memainkannya tetap sama yaitu menggeser-geser petak bernomor acak sampai terbentuk sebuah pola yang teratur, sesuai dengan pola target.

Perpindahan petak-petak tidak bisa dilakukan sembarangan, melainkan harus saling bertukar tempat dengan ruang kosong. Akibatnya, tidak bisa dipastikan bahwa setiap susunan bilangan acak (n x n) - 1 dapat mencapai pola yang diharapkan. Terdapat beberapa kondisi yang harus dipenuhi oleh susunan angka-angka tersebut (akan dibahas lebih lanjut).

Gambar 1. Sliding 15-Puzzle

Permasalahan matematika ini telah menarik perhatian banyak ahli matematika dan programmer dunia. Mereka menciptakan berbagai algoritma yang berkaitan dengan N-Puzzle Problem. Algoritma inipun tersebar dan mudah didapatkan dari buku ataupun internet. Tentu saja keanekaragaman algoritma tersebut memiliki ciri khas yang berbeda-beda, misal dari sisi kompleksitas algoritmanya.

Kompleksitas algoritma adalah besaran yang dipakai untuk menerangkan model abstrak pengukuran waktu dan ruang yang dibutuhkan dalam menjalankan algoritma [1]. Model abstrak tersebut harus independen dari pertimbangan mesin dan compiler apapun. Algoritma yang baik adalah algoritma yang meminimumkan kebutuhan waktu dan ruang.

Dalam makalah ini, penulis menjelaskan sebuah algoritma yang digunakan untuk menentukan solvabilitas sliding N-puzzle. Algoritma tersebut menentukan apakah suatu susunan bilangan acak (n x n) - 1 dapat disusun mencapai target melalui proses pergeseran petak. Jika ya, maka dikatakan bahwa puzzle tersebut *solvable*. Kemudian dari algoritma tersebut akan ditentukan kompleksitas algoritma dari sisi kompleksitas waktunya.

II. N-PUZZLE

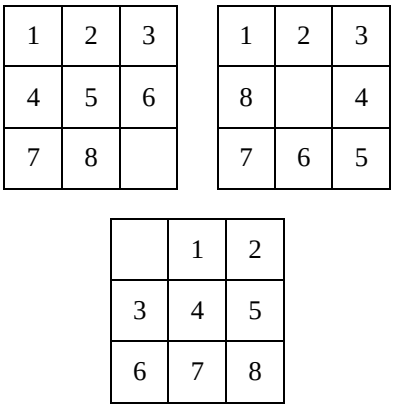
A. Sejarah

N-Puzzle, atau yang dikenal juga sebagai Sliding Puzzle, pertama kali diperkenalkan di New York pada tahun 1879 oleh Noyes Palmer Chapman [2]. Versi pertama N-puzzle memiliki ukuran 4 x 4 petak dan diberi nama Gem Puzzle. Pada tahun 1880 permainan ini menjadi sangat populer di kalangan warga Amerika Serikat, dan menjadi salah satu permainan yang paling digemari disana. Versi Puzzle ini kini dikenal dengan nama 15-Puzzle. Terdapat juga versi puzzle yang lebih kecil dengan ukuran 3 x 3 petak bernama 8-Puzzle.

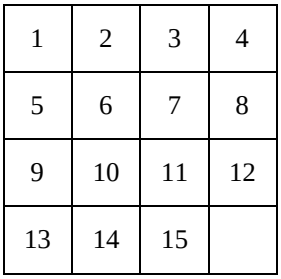
B. Aturan Permainan

N-puzzle terdiri atas *n* x *n* petak, dengan satu petak kosong, sehingga banyaknya petak pada puzzle adalah N, dengan N = (*n* x *n*) - 1. Sejumlah N petak ini kemudian diberi nomor 1 s/d N.

Untuk memecahkan permasalahan N-Puzzle Problem, sebanyak N petak perlu diubah posisinya sehingga membentuk pola yang teratur. Variasi puzzle yang beredar mengharuskan puzzle mencapai pola target yang berbeda-beda pula, namun pada makalah ini akan digunakan versi yang umum yaitu angka berurut dari kiri ke kanan, dari atas ke bawah, dan pojok kanan bawah kosong.



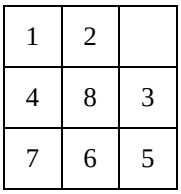
Gambar 2. Berbagai variasi pola target *solved* 8-Puzzle.



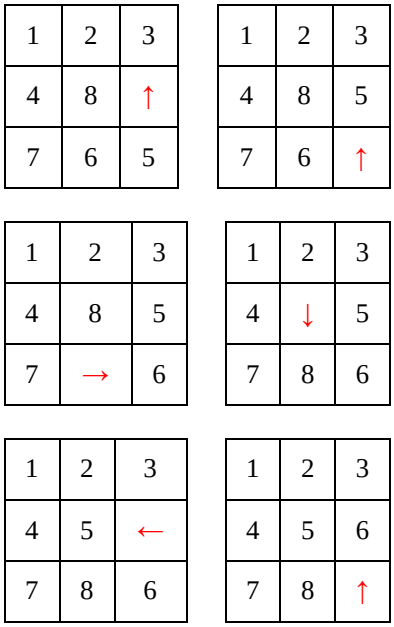
Gambar 3. Pola target *Solved* 15-Puzzle sesuai perjanjian.

Memindahkan petak-petak N-puzzle tidaklah sembarangan seperti bermain puzzle atau jigsaw pada umumnya. Setiap petak dipindahkan dengan aturan digeser, sehingga bertukar posisi dengan ruang kosong. Contohnya pada Gambar 3, petak yang bisa digerakkan hanyalah petak bernomor 15 ke kanan, atau petak nomor 12 ke bawah. Jika ingin memindahkan petak lain, maka perlu dilakukan perubahan posisi petak-petak yang menghalangi. Dengan cara tukar-menukar petak serta ruang kosong sedemikian rupa, N-Puzzle Problem akhirnya bisa dipecahkan (*solved*).

Sebuah susunan bilangan acak dengan jumlah (*n* x *n*) -1 disebut sebagai *instance* dari N-Puzzle. Setiap *instance* dari N-Puzzle dapat ditentukan solvabilitasnya, yaitu menentukan apakah melalui proses penggeseran *instance* dapat mencapai pola target atau tidak.



Gambar 4. Salah satu *instance* dari 8-Puzzle.



Gambar 5. Langkah memecahkan *instance* pada Gambar 4.

Sebagai contoh, diberikan *instance* sesuai pada Gambar 4, maka bisa dimisalkan langkah pertama memindahkan 3 ke atas. Kemudian dilakukan proses pergeseran seperti perputaran dengan arah berlawanan arah jarum jam antara petak 5,6, 8, serta ruang kosong. Karena *instance* terakhir pada Gambar 5 *solved*, maka *instance* Gambar 4 *solvable*.

Dapat dikatakan bahwa terdapat hubungan antara *instance solvable* dengan pola target. *Instance solvable* jika disusun akan dapat mencapai pola target, begitupula sebaliknya pola target jika diacak sesuai aturan akan menghasilkan *instance solvable*. Sehingga, *instance* sisanya dikatakan *unsolvable* karena tidak dapat dicapai melalui proses pengacakan pola target.

Terdapat ciri khas yang bisa dilihat dari hubungan solvabilitas *instance* dengan ukuran puzzle (*n* x *n*) [3].

Pertama, perlu diperhatikan jumlah inversi pada *instance*. Inversi adalah kemunculan b sebelum a, padahal a<b. Kemudian, lihat jumlah baris *n*. Jika *n* ganjil, maka *instance solvable* jika jumlah inversi genap. Jika *n* genap, maka *instance solvable* jika:

- ruang kosong ada di baris genap dari bawah (*second-last, fourth-last,* etc.), dan jumlah inversi ganjil.
- ruang kosong ada di baris ganjil dari bawah (last, *third-last*, *fifth-last*, etc.), dan jumlah inversi genap.

Untuk 8-Puzzle, setiap *instance solvable* bisa dipecahkan dengan 31 langkah, sedangkan untuk 15-Puzzle dipecahkan dengan paling banyak 80 langkah [4], [5].

C. Pembuktian Solvabilitas15-Puzzle

Pada kenyataannya, sejumlah setengah dari seluruh *instance* 15-Puzzle mustahil untuk dipecahkan, baik sebanyak apapun langkah pergeseran dilakukan. Hal ini dibuktikan oleh Johnson & Story pada tahun 1879. Dengan memanfaatkan paritas, diperoleh fungsi dari *instance*s yang bersifat invarian pada paritas permutasi *n* x *n* petak ditambah paritas *taxicab distance* (jumlah baris x jumlah kolom) antara ruang kosong dengan pojok kanan bawah. Sifat invarian ini diperoleh karena setiap langkah pergeseran, kedua paritas berubah sekaligus. Akhirnya diperoleh kesimpulan bahwa sebuah *instance* 15-puzzle bersifat *solvable* jika dan hanya jika permutasi petak yang salah berjumlah genap saat ruang kosong berada di pojok kanan bawah.

Hasil pembuktian tersebut kemudian dikembangkan [6]. Telah didefinisikan inversi adalah adanya angka i yang muncul sebelum n, padahal n < i, dilambangkan n_i. Maka dapat ditulis,

$$N \equiv \sum_{i=1}^{15} n_i = \sum_{i=2}^{15} n_i$$

N = i(p) adalah banyaknya permutasi inversi. Misalkan *e* adalah posisi baris ruang kosong pada puzzle, jika *N + e* berjumlah genap maka *instance solvable*. Jika simbol permutasi $(-1)^{i(p)}$ = +1, maka *instance solvable*, jika simbol permutasi = -1, maka *instance* un*solvable*.

III. KOMPLEKSITAS ALGORITMA

A. Algoritma

Algoritma adalah urutan logis langkah-langkah penyelesaian masalah secara sistematis [1]. Algoritma banyak digunakan dalam matematika dan ilmu komputer untuk penghitungan, pemrosesan data, dan penalaran otomatis. Algoritma terdiri atas instruksi-instruksi yang telah terdefinisi, mulai dari kondisi awal (mungkin menerima *input*), sampai pada kondisi akhir dan menghasilkan *output*.

Algoritma selain harus benar, juga haruslah efisien. Efisiensi suatu algoritma diukur dari jumlah waktu (satuan s atau ms) dan ruang memori (satuan byte atau KB) yang dibutuhkan untuk menjalankan algoritma tersebut. Waktu dan ruang memori yang dibutuhkan oleh sebuah algoritma sangat bergantung pada jumlah data yang diproses. Semakin banyak data, maka waktu dan ruang yang dibutuhkan semakin besar,sehingga performa kurang sesuai dengan yang diharapkan.

Menentukan kompleksitas suatu algoritma tidak bisa hanya didasarkan pada waktu mengeksekusi program. Alasannya adalah karena tiap arsitektur komputer yang berbeda menghasilkan waktu yang berbeda-beda pula dalam melaksanakan operasi-operasi dasar seperti penjumlahan, perkalian, dan perbandingan. Selain itu, algoritma yang diterjemahkan menjadi *executable code* sangat dipengaruhi oleh *compiler*. Berbeda *compiler* maka akan menghasilkan kode tingkat mesin akan menggunakan waktu dan ruang memori yang berbeda pula. Padahal, mengukur kompleksitas waktu dan ruang haruslah tidak terikat oleh arsitektur komputer atau compiler manapun.

Semakin sedikit waktu dan ruang yang dibutuhkan untuk menjalankan algoritma, maka algoritma dapat dikatakan efisien. Kini, orang berlomba-lomba mengembangkan algoritma yang semakin baik dan semakin efisien, sehingga terdapat begitu banyak alternatif dan variasi algoritma untuk suatu tujuan yang sama. Setiap algoritmanya memiliki kekhasan dan kebutuhan waktu serta ruangnya berbeda-beda pula. Oleh karena itu, efisiensi algoritma sering dijadikan pokok utama dalam menentukan pilihan penggunaan algoritma tertentu. Misalnya saat ingin melakukan pencarian array, terdapat alternatif dengan menggunakan *sequential search* atau *binary search*. Kasus lain, misalnya saat mengurutkan elemen, alternatif yang tersedia yaitu *counting sort, bubble sort, insertion sort, selection sort, mergesort, quicksort*, dan sebagainya.

B. Kompleksitas Waktu dan Ruang

Kompleksitas Algoritma terbagi menjadi dua, yaitu kompleksitas waktu dan kompleksitas ruang [1]. Kompleksitas waktu, dinotasikan *T(n)*, diekspresikan sebagai jumlah tahapan komputasi, dengan ukuran masukan *n*. Sedangkan kompleksitas ruang, dinotasikan *S(n)*, diekspresikan sebagai jumlah memori yang digunakan oleh struktur data dengan ukuran masukan n. Akhirnya dapat ditentukan sebuah hubungan antara peningkatan ukuran masukan *n* dengan pertambahan waktu serta ruang memori yang dibutuhkan algoritma.

Pada waktu silam, kompleksitas memori sangat dipertimbangkan dalam menilai suatu algoritma. Hal ini disebabkan mahalnya teknologi yang memungkinkan penyimpanan data dalam jumlah besar pada masa tersebut. Namun sekarang biaya untuk memori relatif murah, sehingga ukuran memori tidak lagi menjadi persoalan yang dianggap penting, sehingga kompleksitas ruang tidak akan dibahas lebih lanjut.

Penyingkatan waktu selamanya akan menjadi kejaran bagi para penmbuat algoritma. Oleh karena itu, perlu

dipelajari mengenai kompleksitas waktu. Kompleksitas waktu menghitung banyaknya operasi yang dilakukan oleh algoritma [1]. Operasi-operasi yang umum terdapat dalam algoritma adalah penjumlahan, perbandingan, pengisian dan pembacaan nilai, pemanggilan prosedur, dan sebagainya.

Beberapa algoritma mempunya operasi dasar masing-masing yang jika diitung bisa menentukan kompleksitas algoritma tanpa perlu menghitung operasi sisanya dalam algoritma tersebut. Misalnya, algoritma pencarian didasari oleh operasi perbandingan. Algoritma pengurutan didasari oleh operasi perbandingan dan operasi pertukaran elemen. Untuk algoritma perkalian, operasi dasarnya adalah operasi penjumlahan dan perkalian.

```
procedure Sum (input a1, a2, ..., an : integer, output r : integer)
Deklarasi
    i : integer
    jumlah : integer
Algoritma
    jumlah ← 0
    i ← 1
    while i ≤ n do
        jumlah ← jumlah + ai
        i ← i+1
    endwhile {i > n}
    r ← jumlah {hasil sum}
```

Gambar 6. Contoh algoritma untuk menghitung penjumlahan dari n buah integer.

Algoritma pada Gambar 6 memiliki beberapa operasi pengisian nilai dan beberapa operasi penjumlahan. Operasi pengisian nilai (operator "←"),

- jumlah ← 0 1 kali
- i ← 1 1 kali
- jumlah ← jumlah + a_i n kali
- i ← i+1 n kali
- r ← jumlah 1 kali,

sehingga total waktunya adalah $t_1 = 3 + 2n$. Sedangkan untuk Operasi penjumlahan (operator "+"),

- jumlah + a_i n kali
- i+1 n kali,

sehingga total waktunya adalah $t_2 = 2n$.

Dengan demikian, kompleksitas waktu algoritma berdasarkan kedua jenis operasi tersebut adalah,

$$T(n) = t_1 + t_2 = 3 + 2n + 2n = 4n + 3.$$

Misalkan n adalah banyaknya data, berdasarkan kasusnya, kompleksitas dibedakan atas tiga macam, yaitu $T_{max}(n)$ yang merupakan kompleksitas waktu untuk kasus terburuk (worst case), $T_{min}(n)$ yang merupakan kompleksitas waktu untuk kasus terbaik (best case), dan $T_{avg}(n)$ yaitu kebutuhan waktu rata-rata yang diperlukan algoritma sebagai fungsi dari n. Ketiganya mungkin tidak terlihat pada algoritma Gambar 6, tetapi untuk algoritma lain seperti algoritma pencarian atau algoritma pengurutan, ketiga notasi $T(n)$ tersebut menjadi pertimbangan yang amat penting.

C. Kompleksitas Waktu Asimptotik

Untuk mengetahui pertumbuhan $T_{min}(n)$ dan $T_{max}(n)$ bersamaan dengan meningkatnya ukuran masukan, digunakan tiga notasi kompleksitas waktu asimptotik, yaitu Big-O, Big-Omega, dan Big-Tetha.

$T(n) = O(f(n))$ artinya $T(n)$ berorde paling besar f(n), bila terdapat konstanta C dan n_0 maka berlaku $T(n) \leq C.\ f$(n), untuk $n \geq n_0$.

$T(n) = \Omega(g(n))$ artinya $T(n)$ berorde paling kecil $g(n)$, bila terdapat konstanta C dan n_0 maka berlaku $T(n) \geq C.\ f$(n), untuk $n \geq n_0$.

$T(n) = \Theta(h(n))$ artinya $T(n)$ berorde sama dengan $h(n)$ jika $T(n) = O(h(n))$ dan $T(n) = \Omega(g(n))$.

Tiap-tiap algoritma memnpunyai kompleksitas waktu asimptotik masing-masing [1]. Untuk spektrum kompleksitas waktu algoritma adalah:

$O(1) < O(\log n) < O(n) < O(n \log n) < O(n^2) < O(n^3) <$ ...

(polinomial)

$< O(2^n) < O(n!)$.

(eksponensial)

Jika $O(f(n))$ muncul sebelum $O(g(n))$ $(f(n) \leq g(n))$ berarti algoritma dengan $O(f(n))$ lebih efisien.

Kelompok Algoritma	Nama
$O(1)$	konstan
$O(\log n)$	logaritmik
$O(n)$	lanjar
$O(n \log n)$	n log n
$O(n^2)$	kuadratik
$O(n^3)$	kubik
$O(2^n)$	eksponensial
$O(n!)$	faktorial

Gambar 7. Kelompok algoritma berdasarkan kompleksitas waktu asimptotiknya.

Algoritma	$T_{min}(n)$	$T_{avg}(n)$	$T_{max}(n)$
Quicksort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$
Mergesort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n \log n)$
Bubble sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Insertion Sort	$\Omega(n)$	$\Theta(n^2)$	$O(n^2)$
Selection Sort	$\Omega(n^2)$	$\Theta(n^2)$	$O(n^2)$
Counting sort	$\Omega(n+k)$	$\Theta(n+k)$	$O(n+k)$
Tree sort	$\Omega(n \log n)$	$\Theta(n \log n)$	$O(n^2)$

Gambar 8. Beberapa kompleksitas waktu algoritma pengurutan.

D. Teorema Big-O

Misalkan $T_1(n) = O(f(n))$ dan $T_2(n) = O(g(n))$, maka berlaku [1],

- $T_1(n) + T_2(n) = O(max(f(n), g(n))$
- $T_1(n) + T_2(n) = O(f(n) + g(n))$
- $T_1(n)T_2(n) = O(f(n))O(g(n)) = O(f(n)g(n))$
- $O(cf(n)) = O(f(n))$, c adalah konstanta
- $f(n) = O(f(n))$.

IV. ALGORITMA MENENTUKAN SOLVABILITAS N-PUZZLE

Diberikan sebuah *instance* N-puzzle. Algoritma yang diimplementasikan dalam bahasa pemrograman C++ berikut akan melakukan pengecekan untuk menentukan solvabilitas *instance* tersebut, apakah *solvable* atau un*solvable* [3]. Program dibawah ini bersifat umum dan berlaku untuk N-Puzzle dengan ukuran minimum 3 x 3.

Program Solvability

```
#include <iostream>
#define n 4                          //n = 4 berarti 15-Puzzle
using namespace std;

int getInvCount(int arr[])
//Menghitung inversi
{
   int count = 0;
   for (int i = 0; i < n * n - 1; i++)
      for (int j = i + 1; j < n * n; j++)
        if (arr[j] && arr[i] && arr[i] > arr[j])
          count++;
   return count;
}

int findXPosition(int puzzle[n][n])
//Menghitung posisi baris ruang kosong dari bawah
{
  for (int i = n - 1; i >= 0; i--)
     for (int j = n - 1; j >= 0; j--)
       if (puzzle[i][j] == 0)
         return N - i;
}

bool isSolvable(int puzzle[n][n])
{
    int count = getInvCount((int*)puzzle);
    // Untuk n ganjil, return true jika inversinya genap
   if (n & 1)
      return !(invCount & 1);
  else{     // n genap
      int pos = findXPosition(puzzle);
      if (pos & 1)
         return !(invCount & 1);
      else
         return invCount & 1;}
}
```

```
/* Driver program */
int main()
{
    //Input instance N-Puzzle
    //0 menandakan ruang kosong
    int puzzle[n][n] =
   {
      {12, 1, 10, 2},
      {7, 11, 4, 14},
      {5, 0, 9, 15},
      {8, 13, 6, 3},
   };

 //Contoh instance lain
 /*
   int puzzle[n][n] = {
              {1, 8, 2},
              {0, 4, 3},
              {7, 6, 5}};

   int puzzle[n][n] = {
              {13, 2, 10, 3},
              {1, 12, 8, 4},
              {5, 0, 9, 6},
              {15, 14, 11, 7},
            };

   int puzzle[n][n] = {
              {6, 13, 7, 10},
              {8, 9, 11, 0},
              {15, 2, 12, 5},
              {14, 3, 1, 4},
            };


   int puzzle[n][n] = {
              {3, 9, 1, 15},
              {14, 11, 4, 6},
              {13, 0, 10, 12},
              {2, 7, 8, 5},
            };
   */

   //Output
   isSolvable(puzzle)? cout << "Solvable":
                       cout << "Not Solvable";
   return 0;
}
```

Pada driver atau program utama $T_1(n) = n^2 + 2$

- Operasi Pengisian Matriks
  Operasi pengisian nilai dilakukan sebanyak n x n
  $t_1 = n^2$.
- Operasi Output Solvabilitas
  Operasi perbandingan dilakukan 1 kali
  Operasi pencetakan ke layar dilakukan 1 kali
  $t_2 = 1 + 1 = 2$.

Pada fungsi getInvCount

- Operasi perbandingan dilakukan sebanyak *n* x *n*
$$t_3 = n^2.$$
- Operasi pengisian nilai sebanyak n x n + 1
$$t_4 = n^2 + 1.$$
- Operasi penjumlahan dilakukan sebanyak *n* x *n*
$$t_5 = n^2.$$

Pada fungsi findXPosition

- Operasi perbandingan dilakukan sebanyak *n* x *n*
$$t_6 = n^2.$$

Pada fungsi is*Solvable*

- Operasi pengisian nilai memanggil getInvCount
$$t_7 = t_3 + t_4 + t_5 = 3n^2 + 1$$
- Operasi percabangan pertama
$$t_8 = 1$$
- Operasi pengisian nilai memanggil findXPosition
$$t_9 = t_6 = n^2$$
- Operasi percabangan kedua
$$t_{10} = 1$$

Jika n ganjil maka tidak masuk ke percabangan kedua sehingga kebutuhan waktunya,

$$T_2(n) = t_7 + t_8 = 3n^2 + 2.$$

Jika n genap maka semua kode dieksekusi sehingga kebutuhan waktunya,

$$T_2(n) = t_7 + t_8 + t_9 + t_{10} = 4n^2 + 3$$

Sehingga untuk keseluruhan program diperoleh kebutuhan waktunya

$$T(n) = T_1(n) + T_2(n)$$

$T(n) = (n^2 + 2) + (3n^2 + 2) = 4n^2 + 4$, $O(n^2)$, *n* ganjil

atau

$T(n) = (n^2 + 2) + (4n^2 + 3) = 5n^2 + 5$, $O(n^2)$, *n* genap

V. KESIMPULAN

N-Puzzle Problem adalah sebuah permasalahan matematika, berukuran n x n dengan jumlah petak 1 petak kosong dan N petak bernomor (N = n x n -1). Cara memecahkan permasalahan tersebut yaitu dengan menggeser petak sampai terbentuk pola target. Sebuah susunan acak dari n x n -1 petak disebut *instance*. Inversi adalah adanya kemunculan bilangan b sebelum a padahal a < b. Sebuah *instance* N-Puzzle dapat ditentukan solvabilitasnya. Jika n ganjil *solvable* jika jumlah inversinya genap. Jika n genap, syarat agar *solvable* adalah ruang kosong terletak baris ganjil dari bawah tetapi jumlah inversi genap, atau ruang kosong terletak di baris genap dari bawah tetapi jumlah inversinya ganjil.

Algoritma yang baik adalah algoritma yang efisien, artinya kebutuhan waktu dan ruang memorinya semininimum mungkin. Sebuah algoritma bisa ditentukan kompleksitas algoritmanya baik dari sisi waktu dengan notasi T(n) ataupun ruang memori S(n). Kebutuhan waktu terbagi menjadi $T_{min}(n)$, $T_{max}(n)$, dan $T_{avg}(n)$. Terdapat pula notasi kompleksitas waktu asimptotik yang digunakan dalam membanding-bandingkan performa tiap algoritma, yaitu Big-O $O(f(n))$, Big-Omega $\Omega(g(n))$, dan Big-Tetha $\Theta(h(n))$.

. Algoritma yang diimplementasikan dalam bahasa pemrograman C++ melakukan pengecekan untuk menentukan solvabilitas *instance* tersebut, apakah *solvable* atau un*solvable*. Diperoleh kompleksitas algoritmanya $O(n^2)$. Hasil kebutuhan waktunya adalah $T(n) = 4n^2 + 4$, n ganjil, atau $T(n) = 5n^2 + 5$, n genap.

VI. PENUTUP

Puji syukur kepada Tuhan Yang Maha Kuasa karena berkat-Nya yang melimpah sehingga penulis dapat menyelesaikan makalah ini dengan baik. Ucapan terima kasih juga penulis sampaikan kepada kedua orang tua serta teman-teman yang terus memberikan dukungan baik secara moral maupun doa. Penulis turut mengucapkan terima kasih kepada Bapak Rinaldi Munir, selaku dosen dari mata kuliah Matematika Diskrit. Akhir kata, penulis memohon maaf atas ketidaksempurnaannya makalah ini. Penulis berharap makalah ini dapat bermanfaat dan dapat terus dikembangkan sehingga menjadi semakin baik lagi.

REFERENCES

[1] R. Munir, *Matematika Diskrit*, 3rd ed. Bandung: Penerbit INFORMATIKA Bandung, 2010, ch. 10.

[2] J. Slocum, D. Sonneveld, *The 15 Puzzle: How It Drove the World Crazy*. Slocum Puzzle Foundation, 2006.

[3] http://www.geeksforgeeks.org/check-*instance*-15-puzzle-*solvable*/, diakses pada 9 Desember 2016, 04:11 WIB.

[4] Dudeney, H. E, Problem 253 in "The Canterbury Puzzles and Other Curious Problems", 7th ed. London: Thomas Nelson and Sons, 1949.

[5] Brüngger, A.; Marzetta, A.; Fukuda, K.; and Nievergelt, J. *The Parallel Search Bench ZRAM and Its Applications.*

[6] http://mathworld.wolfram.com/15Puzzle.html/, diakses pada 9 Desember 2016, 04:31 WIB.

SUMBER GAMBAR

1 http://entertainment.howstuffworks.com/puzzles/sliding-puzzles.htm

7 R. Munir, *Matematika Diskrit*, 3rd ed. Bandung: Penerbit INFORMATIKA Bandung, 2010, ch. 10.

8 http://bigocheatsheet.com/

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, 9 Desember 2016

Audry Nyonata, 13515087