Penggunaan Algoritma Backtrack dan Aturan Warnsdorff Untuk Menyelesaikan Knight's Tour Problem

Ali Akbar - 13514080
Program Studi Teknik Informatika
Sekolah Teknik Elektro dan Informatika
Institut Teknologi Bandung, Jl. Ganesha 10 Bandung 40132, Indonesia
aliakbr@students.itb.ac.id

***Abstrak*—*Knight's Tour* adalah permasalahan unik di dunia matematika. Kita diminta untuk menemukan urutan langkah bagaimana sebuah kuda pada papan catur dapat mengunjungi seluruh kotak pada papan maksimal sekali. Untuk menyelesaikan permasalahan ini dengan computer kita dapat menggunakan algoritma *backtracking*. Namun algoritma *backtracking* belum efisien dalam memilih langkah-langkah yang menuju ke solusi. Dengan bantuan aturan Warnsdoff untuk pemilihan langkah selanjutnya algoritma *backtracking* menjadi lebih mangkus dibanding sebelumnya.**

Kata Kunci*— *Knight's Tour, backtrack,* Aturan *Warnsdoff

I. PENDAHULUAN

Knight's Tour problem adalah masalah di dunia matematika tentang bagaimana caranya sebuah kuda di atas papan catur 8x8 mengunjungi setiap kotak pada papan maksimal sekali (artinya dalam sekali jalan). Banyak mitos bilang jika permasalahan ini telah digambarkan pada puisi berbahasa sansekerta yang bernama "turagapadabandha" yang artinya "susunan langkah kuda". Namun pastinya permasalahan ini dibawa dan dikaji pertama kali oleh seorang matematikawan bernama Leonhard Euler. Prosedur pertama yang menyelesaikan permasalahan ini dibuat oleh H.C Von Warnsdorf yang dikenal dengan Aturan Warnsdorf.

Banyak cara untuk menyelesaikan permasalahan ini dalam program computer. Misalnya secara naif kita bisa menggunakan *brute force.* Pada *brute force* kita membuat seluruh kemungkinan langkah kuda pada papan. Untuk semua kemungkinan itu dicari mana yang merupakan solusi dari permasalahan. Metode tersebut tidaklah mangkus karena banyak langkah-langkah yang tidak perlu dibangkitkan tapi tetap dibangkitkan.

Salah satu cara yang lebih mangkus untuk menyelesaikan permasalahan ini adalah algoritma *backtracking*. Pada algoritma *backtracking* langkah-langkah solusi ditemukan dengan cara mengiterasi langkah-langkah hingga solusi ditemukan. Jika solusi tidak ditemukan maka kembali ke langkah sebelumnya dan coba kemungkinan langkah lain. Jika seluruh kemungkinan telah dicoba dan tetap tidak menghasilkan solusi maka kembali ke langkah sebelumnya dan coba kemungkinan lain. Karena kita kembali ke keadaan sebelumnya maka algoritma ini disebut dengan algoritma *backtracking*.

Untuk menyelesaikan *knight's tour* dengan *backtracking* maka perlu dibuat fungsi pembangun kemungkinan langkah dan fungsi iterative (atau rekursif) untuk terus memilih langkah selanjutnya. Diperlukan juga proses kembali ke langkah sebelumnya apabila tidak ditemukan solusi. Namun jika fungsi pemilih kemungkinan langkah hanya memilih langkah secara acak maka akan dicoba langkah-langkah yang tidak menuju ke solusi. Maka dari itu untuk mengoptimalkan algoritma *backtracking* kita menggunakan aturan Warnsdoff dalam memilih langkah selanjutnya yang akan dipilih.

Aturan Warnsdoff membuat pemilihan langkah dalam algoritma *backtracking* lebih cerdas. Dalam aturan Warnsdoff untuk menemukan solusi langkah yang harus dipilih pilihlah langkah yang memiliki total langkah selanjutnya paling kecil. Jika ada 2 langkah yang sama maka dicoba keduanya. Jika salah satunya tidak memberikan solusi maka dilakukan *backtracking* ke langkah sebelumnya.

Dalam makalah ini penulis akan membuat analisis dan implementasi pada sebuah program *console* sederhana yang ditulis dalam bahasa java untuk membandingkan penggunaan algoritma *backtracking* biasa dan dengan modifikasi aturan Warnsdoff.

II. LANDASAN TEORI

A. Backtracking

Algoritma *backtracking* adalah algoritma dimana semua kemungkinan langkah selanjutnya dibangun pada state tersebut dan jika tidak ada langkah yang mungkin atau solusi tidak ditemukan pada state ini maka kembali ke state sebelumnya dan ubah ke kemungkinan langkah yang lain.

Properti umum dari algoritma backtrack adalah :

a. Solusi persoalan
   Solusi dapat dinyatakan sebagai vector dengan n-tuple.
b. Fungsi pembangkit
   Dinyatakan sebagai T(k). T(k) membangkitkan vector solusi.
c. Fungsi pembatas
   Dinyatakan sebagai B(x). Fungsi mengembalikan nilai true jika langkah x dapat mengarah ke solusi atau tidak. Jika tidak maka kembali ke langkah sebelumnya.

Kita dapat membuat sebuah pohon urutan langkah-langkah untuk memudahkan kita mencari kemungkinan langkah yang lain. Apabila pohon digunakan, prinsip dari backtracking adalah sebagai berikut [1] :

a. Solusi dicari dengan membentuk lintasan dari akar ke daun. Aturan pembentukan yang dipakai adalah mengikuti metode pencarian mendalam (*DFS*). Simpul yang dilahirkan dinamakan simpul hidup dan simpul yang sedang dibangkitkan dinamakan simpul-E (*Expand-node*). Simpul dinomori sesuai urutan hidupnya
b. Tiap kali simpul-E diperluas, lintasan dibangun olehnya bertambah panjang. Jika lintasan yang sedang dibentuk tidak mengarah ke solusi, maka simpul-E tersebut "dibunuh" sehingga menjadi simpul mati (*dead node*). Fungsi yang digunakan untuk membunuh simpul-E disebut juga fungsi pembatas (*bounding function*). Simpul yang mati tidak perlu diperluas lagi.
c. Jika pembentukan lintasan berakhir dengan simpul mati maka proses pencarian diteruskan dengan membangkitakn simpul anak yang lainnya.Bila tidak ada lagi simpul anak yang dibangkitkan, maka pencarian solusi dilanjutkan dengan runut-balik ke simpul hidup terdekat (simpul orang tua). Selanjutnya simpul ini menjadi simpul-E yang baru. Lintasan baru dibangun kembali sampai lintasan tersebut membentuk solusi.
d. Pencarian dihentikan bila kita telah menemukan solusi atau tidak ada lagi simpul hidup untuk runut-balik.

Pada implementasinya algoritma *backtracking* dapat digunakan secara iterative atau rekursif. Untuk skema umumnya (ditulis dalam pseudocode) adalah sebagai berikut :

a. Versi Rekursif

```
Procedure backtrack(input problem)

Algoritma :
   Untuk setiap kemungkinan langkah pada state tersebut lakukan
        Cek apakah langkah aman
        Jika aman maka panggil backtrack(rest_of_problem)
        Jika salah kembali ke langkah sebelumnya dan cek langkah yang lain.
   Akhir pengulangan.
   Jika tidak ada langkah yang bisa degenerate maka kembalikan tidak ada solusi.
```

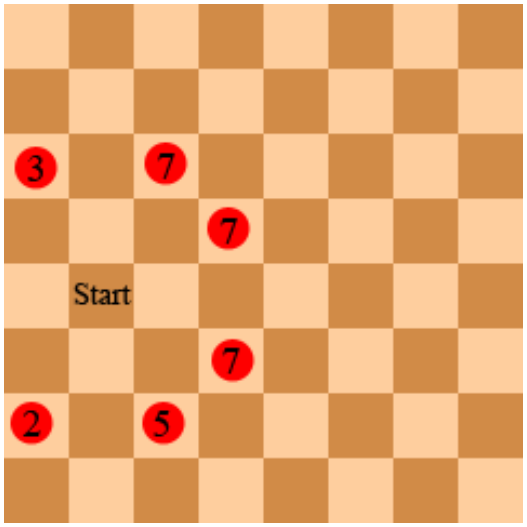
b. Versi Iteratif

```
Procedure backtrack()

Algoritma :
    k = 0
    Selama belum ditemukan solusi
      Cek apakah kemungkinan[k] aman
      Jika aman maka lanjutkan ke k+1
      Jika tidak aman maka kembali ke
      Kemungkinan ke k-1 dan cari
      kemungkinan lanjutan yang lain.
```

B. Aturan Warnsdoff

Pada aturan Warnsdoff langkah yang dapat mengarah ke solusi adalah langkah yang memiliki total langkah selanjutnya paling kecil dibandingkan langkah yang lain. Untuk mengilustrasikannya lihat gambar dibawah.



Pada gambar di atas start adalah posisi kuda saat ini. Tanda merah adalah kemungkinan langkah dan angka pada tanda merah adalah banyaknya langkah selanjutnya yang mungkin jika langkah tersebut diambil. Dari kemungkinan langkah diatas kita cari langkah yang banyak kemungkinan langkah selanjutnya paling kecil yaitu 2. Maka dari start kita pergi ke kotak yang berisi angka 2.

III. IMPLEMENTASI ALGORITMA PADA PROGRAM

A. Algoritma Bactrack

Pada implementasi ini penulis menulis program menggunakan bahasa java (program ditulis dalam bentuk pseudo-code). Pada desainnya penulis membuat 2 struktur data penting yaitu Kelas Tuple dan Kelas KnightTour untuk wrap tuple-tuple dalam satu masalah. Kelas Tuple memiliki struktur dasar yaitu

```
Public class Tuple{
    private int value1;
    private int value2;
    public boolean[][] Passed;
    public Vector<Tuple> nextMove;
    public Tuple PositionBefore;
    /* diteruskan dengan method
setter getter */
 }
```

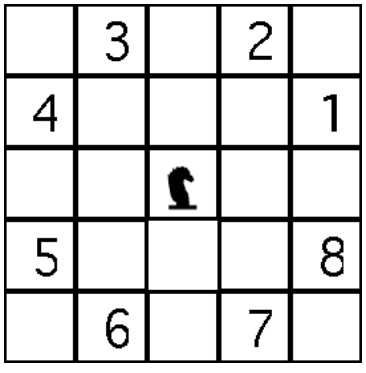
Kelas Tuple menyimpan posisi dalam bentuk (x,y), langkah-langkah selanjutnya yang mungkin akan dicoba pada posisi itu (disimpan dalam nextMove), tuple posisi sebelumnya (disimpan dalam PositionBefore) dan posisi-posisi yang telah dilewati oleh state ini (disimpan dalam array of array of Boolean bernama Passed). Nilai dari nextMove akan diisi oleh fungsi pembangkit (pada kelas wrapper bernama KnightTour).

Selanjutnya dibuat sebuah kelas *wrapper* yang membungkus masalah dan berisi penyelesaian masalah. Kelas ini dinamai KnightTour. Atribut dasarnya adalah sebagai berikut

```
public class KnightTour {
    int[][] Mem;
    boolean[][] Finished;
    int assignment;
    Tuple position;
    Vector<Tuple> solution;
     /* dilanjutkan dengan ethod
pembangun */
 }
```

Mem berguna untuk menyimpan urutan langkah setelah solusi digenerate. Finished berfungsi untuk melihat apakah semua sudah terisi atau belum (solusi sudah ditemukan atau belum, bukan untuk pembanding apakah telah dilalui sebelumnya). Assignment berisi banyaknya langkah yang dilakukan. Position menyimpan posisi saat ini. Solution adalah solusi urutan langkah-langkah dalam bentuk tuple.

Untuk membangun langkah-langkah yang dapat dilakukan pada sebuah posisi kita buat sebuah prosedur bernama findPath. findPath berguna untuk mengisi nextMove pada tuple. Terdapat 8 kemungkinan langkah saat kuda berada pada papan catur. 8 kemungkinan langkah diilustrasikan pada gambar dibawah :



Namun perlu kita cek apakah posisi langkah selanjutnya mungkin (misal ada di 0,2 maka tidak mgkin bisa ke (-1,0) walaupun masuk ke 8 kemungkinan) dan langkah selanjutnya belum pernah dilalui oleh state itu. Pseudo code untuk fungsi ini adalah :

```
Public void findPath(){
 For kemungkinan 1 sampai 8
    /* x dan y adalah posisi
kemungkinan ke i*/
    If (Passed[x][y] == false) and
(possible) then

nextMove.addElement(kemungkinan ke-i)
    Else do nothing
 }
```

Setelah itu kita buat prosedur pengecek apakah dimungkinkan ada langkah atau tidak. Prosedur pengecek ini dinamai isMovePossible().

```
 public boolean isMovePossible(){
return(!position.nextMove.isEmpty());
     }
```

Setelah itu buat fungsi untuk mengecek apakah solusi telah ditemukan atau belum.

```
public boolean AllFinished(){
      boolean a = true;
      for (int i = 0; i < 8; i++){
         for (int j = 0; j < 8; j++){
           if (Finished[i][j] == false)
              a = false;
         }
      }
      return a;
   }
```

Setelah semua method-method pembangun kita buat sekarang kita buat proses untuk menyelesaikan Knight's Tour Problem secara backtrack. Pada proses ini pertama kita tandai bahwa posisi yang kita tempati sekarang telah dilewati. Selanjutnya selama masih ada kotak yang belum ditemukan solusinya maka kita lakukan skema dasar algoritma *backtrack*. Kita simpan langkah sekarang pada PositionBefore lalu kita isi seluruh informasi (Tuple) untuk langkah sekarang dengan nextMove indeks ke 0. Apabila tidak terdapat nextMove (tidaka ada lagi kemungkinan jalan namun semua belum diselesaikan) maka kembali ke langkah sebelumnya namun langkah yang telah kita coba ditandai sebagai sudah dilewati dengan cara membuat Passed[i][j] (i dan j adalah posisi sekarang) bernilai true untuk tuple itu. Untuk lebih jelasnya dapat dilihat pada source code di bawah ini :

```
public void Process(){
      /* Menandai posisi pertama sudah dilewati */
      int i, j;
      i = position.getValue1();
      j = position.getValue2();
```

```
        Finished[i][j] = true;
        PositionBefore.Passed[i][j] = true;
        int count = 0;
        Tuple PositionBefore = new Tuple();

        /* Iterasi ke langkah selanjutnya */
        while (AllFinished() == false){
           /* Mengconstruct seluruh jalan yang mungkin*/
           findPath();

           /* Menghitung berapa pengecekan ( langkah ) */
           count++;

           /* Jika mungkin maju ke sel selanjutnya */
           if (isMovePossible()){
              /* Catat posisi sebelumnya */
              i = position.getValue1();
              j = position.getValue2();
              PositionBefore.setValue(i, j);
              PositionBefore.nextMove =
position.nextMove;
              PositionBefore.Passed = position.Passed;

              /* Maju ke posisi berikutnya */
              Tuple T1 =
PositionBefore.nextMove.remove(0);
              position.setValue(T1.getValue1(),
T1.getValue2());
              position.nextMove = T1.nextMove;
              position.Passed = T1.Passed;
              i = position.getValue1();
              j = position.getValue2();
              Finished[i][j] = true;
              Position.PositionBefore = PositionBefore;
              /*tambahkan ke kemungkinan solusi*/
              solution.addElement(position);

           }
           /* Jika tidak kembali ke posisi sebelumnya */
           else{
              /* hapus dari kemungkinan solusi */
              Tuple Trash =
solution.remove(solution.size()-1);
              i = position.getValue1();
              j = position.getValue2();
              Finished[i][j] = false;
              /* kembali ke posisi sebelumnya*/
              position.PositionBefore.Passed[i][j] = true;
              position = position.PositionBefore;
           }
        }
        assignment = count++;
     }
  }
```

Prosedur proses() menjalankan algoritma backtracking secara iterative dengan memanfaatkan struktur data yang telah dibuat. Langkah yang diambil pertama kali adalah langkah yang berada di indeks 0 pada vector nextMove di tuple position. Program utama berisi inisialisasi posisi awal kuda dan menampilkan urutan langkah pada matriks. Untuk menampilkan urutan langkah digunakan fungsi printSolution() yang berguna untuk memberi label pada matriks sesuai dengan urutan tuple pada solution.

```
  public void printSolution(){
        int i, j, k;
        for (i = 0; i < 64; i++){
          j = solution.get(i).getValue1();
          k = solution.get(i).getValue2();
          Mem[j][k] = i;
        }
        for (j = 0; j < 8; j++){
           for (k = 0; k < 8; k++){
              System.out.print(Mem[j][k]+"\t");
           }
           System.out.println();

           }
     }
```

Program utama berisi :

```
  public static void main(String[] args) {
        KnightTour K = new KnightTour();
        K.setPosition(0,0);
        K.Process();
        K.printSolution();
        System.out.println("total move :"+K.totalMove());
     }
```

B. Modifikasi dengan Warnsdoff

Aturan warnsdoff dimasukkan sebagai tambahan pada findPath. Setelah findPath dilakukan dilakukan sortPath untuk mengurutkan langkah berdasarkan langkah selanjutnya dimulai dari nilai yang terkecil. Pseudo code untuk algoritma sort adalah sebagai berikut

```
  public void sortPath(){
        Tuple temp;
        int idxMin;
        for (int i = 0; i < 8; i++){
           idxMin = i;
           for (int j = i + 1; i < 8; i++){
              if (total nextMove pada
position.nextMove[idxMin] > nextMove[j])
                 idxMin = j;
           }
           /* swap */
           temp = positon.nextMove[i];
           positon.nextMove[i] =
positon.nextMove[idxMin];
           positon.nextMove[idxMin] =
positon.nextMove[i];
        }
     }
```

Pemilihan algoritma sorting yang lebih bagus akan membuat program berjalan lebih cepat. Namun tidak

mengurangi banyak langkah yang dilalui oleh kuda. Sortpath ditambahkan pada bagian process() di kelas KnightTour. Tepatnya sortPath dipanggil setelah adanya findPath. Berikut adalah process() yang baru setelah adanya sortPath.

```
public void Process(){
    /* Menandai posisi pertama sudah dilewati */
    int i, j;
    i = position.getValue1();
    j = position.getValue2();
    Finished[i][j] = true;
    int count = 0;
    Tuple PositionBefore = new Tuple();

    /* Iterasi ke langkah selanjutnya */
    while (AllFinished() == false){
      /* Mengconstruct seluruh jalan yang mungkin*/
      findPath();

      /* Mengurutkan Path */
      sortPath()

      /* Menghitung berapa pengecekan ( langkah ) */
      count++;

      /* Jika mungkin maju ke sel selanjutnya */
      if (isMovePossible()){
        /* Catat posisi sebelumnya */
        i = position.getValue1();
        j = position.getValue2();
        PositionBefore.setValue(i, j);
        PositionBefore.nextMove =
position.nextMove;
        PositionBefore.Passed = position.Passed;
        PositionBefore.Passed[i][j] = true;
        Position.PositionBefore = PositionBefore;


        /* Maju ke posisi berikutnya */
        Tuple T1 =
PositionBefore.nextMove.remove(0);
        position.setValue(T1.getValue1(),
T1.getValue2());
        position.nextMove = T1.nextMove;
        position.Passed = T1.Passed;
        i = position.getValue1();
        j = position.getValue2();
        Finished[i][j] = true;
        /*tambahkan ke kemungkinan solusi*/
        solution.addElement(position);

      }
      /* Jika tidak kembali ke posisi sebelumnya */
      else{
        /* hapus dari kemungkinan solusi */
        Tuple Trash =
solution.remove(solution.size()-1);
        i = position.getValue1();
        j = position.getValue2();
        Finished[i][j] = false;
        /* kembali ke posisi sebelumnya*/
        Position.PositionBefore.Passed[i][j] = true;
        position = position. PositionBefore;
      }
    }
    assignment = count++;
  }
}
```

Untuk program utama masih sama seperti pada bagian a dimana kuda diinisialisasi di titik 0,0 lalu diselesaikan dengan Process dan diprint hasil dan banyak langkahnya.

III. HASIL PENGUJIAN

Pengujian dilakukan dengan input kuda berada titik 0,0 saat program dimulai. Jika dijalankan dengan algoritma backtrack biasa maka akan menghasilkan output sebagai berikut :

```
1       60      39      34      31      18      9       64
38      35      32      61      10      63      30      17
59      2       37      40      33      28      19      8
36      49      42      27      62      11      16      29
43      58      3       50      41      24      7       20
48      51      46      55      26      21      12      15
57      44      53      4       23      14      25      6
52      47      56      45      54      5       22      13

total move : 8250732
```

Pada hasil ditampilkan matriks yang menunjukkan langkah-langkah untuk mencapai solusi dan banyak langkah yang dicoba untuk mencapai solusi. Total move pada algortima backtrack biasa ialah 8250732 sementara apabila menggunakan modifikasi dengan bantuan aturan wansdorf hasilnya sebagai berikut adalah sebagai berikut :

```
1       4       57      20      41      6       43      22
34      19      2       5       58      21      40      7
3       56      35      60      37      42      23      44
18      33      48      53      46      59      8       39
49      14      55      36      61      38      45      24
32      17      52      47      54      27      62      9
13      50      15      30      11      64      25      28
16      31      12      51      26      29      10      63

total move :366
```

Total move nya hanya 366. Berarti dapat kita simpulkan memang aturan warnsdoff ditambah dengan backtrack sangat efektif untuk menyelesaikan permasalahan Knight's Tour dalam hal banyak langkah.

IV. KESIMPULAN

Algoritma *backtrack* mampu menghasilkan solusi dari permasalahan Knight's Tour. Namun pada algoritma *backtrack* masih dilalui langkah-langkah yang tidak mengarah ke solusi walaupun itu lebih sedikit dari algoritma brute force. Akibatnya algoritma *backtrack*

belum cukup efisien dalam hal langkah percobaan. Untuk membuatnya lebih efisien kita dapat memodifikasi fungsi pembatas algoritma *backtrack* dengan bantuan aturan Warnsdoff. Dengan demikian kita bisa memangkas langkah-langkah yang tidak perlu. Aturan Warnsdoff berbunyi langkah yang mengarah ke solusi adalah langkah yang langkah setelahnya paling minimum dibanding langkah yang lain. Untuk menemukan langkah selanjutnya yang paling minimum kita dapat melakukan sorting daftar kemungkinan langkah sebelum memilih langkah. Pemilihan metode sorting mempengaruhi kecepatan komputasi namun tidak mengubah banyak langkah yang dicoba.

V. SARAN

a. Untuk pemogram yang ingin mengimplementasikan ulang diharap menggunakan Bahasa yang lebih advanced atau library yang lebih advanced sehingga tidak perlu membuat kelas Tuple pada program. (Seperti struktur tuple pada C++ 11)
b. Pada program urutan langkah dibuat setelah solusi digenerate padahal urutan langkah bisa dibuat pada saat algoritma mencari jalan yang tepat menuju solusi dijalankan.
c. Langkah sebelumnya tidak perlu disimpan pada Tuple dalam matriks berukuran 8x8 karena membuat program memakan banyak. Langkah sebelumnya bisa disimpan dalam vector dinamis yang ukurannya jauh lebih kecil (karena maksimal hanya ada 7 langkah predesor).
d. Gunakan algoritma sorting yang paling mangkus sehingga waktu komputasi menjadi lebih cepat.

VI. UCAPAN TERIMAKASIH

Penulis mengucapkan puji dan syukur kepada Allah SWT karena berkat rahmatnya penulis dapat menyelesaikan tulisan ini. Penulis juga berterimakasih kepada dosen pengajar IF2211 Strategi Algoritma Bapak Ir.Rinaldi Munir dan Ibu Nur Ulfa Mauladevi karena berkat pengajarannya penulis memiliki pengetahuan tentang algoritma *backtracking* yang menjadi dasar pembuatan makalah ini.

Penulis juga berterima kasih atas dukungan teman-teman dalam penyelesaian tulisan ini.

REFERENCES

[1] Munir, Rinaldi, *Diktat Kuliah IF 2120, , Strategi Algoritma, Program Studi Teknik Informatika, STEI ITB,* 2006, pp. VIII-2 – VIII-18, IX-1 – IX-9.
[2] http://www.mactech.com/articles/mactech/Vol.14/14.11/TheKnightsTour/index.html (diakses 7 Mei 2016)
[3] http://www.tri.org.au/knightframe.html (diakses 7 mei 2016).

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, 7 Mei 2016

Ali Akbar, 13514080