PENERAPAN DAN OPTIMASI ALGORITMA RUNUT-BALIK ITERATIF PENCARIAN SOLUSI PADA PERMAINAN *SUDOKU*

Monterico Adrian [13505036]

Program Studi Teknik Informatika, Institut Teknologi Bandung
Jalan Ganesha 10, Bandung
e-mail: if15036@students.if.itb.ac.id

ABSTRAK

Runut-balik (*backtracking*) adalah algoritma yang berbasis pada *DFS* (*Depth First Search*) untuk mencari solusi persoalan secara lebih mangkus dan sistematis dan merupakan perbaikan dari algoritma *brute-force*. Dengan runut-balik, kita tidak perlu memeriksa semua kemungkinan solusi yang ada karena pencarian ke arah solusi saja yang selalu dipertimbangkan. Akibatnya, waktu pencarian dapat dihemat. Ada dua versi algoritma runut-balik yaitu rekursif dan iteratif. Runut-balik lebih alami dinyatakan dalam versi rekursif. Tetapi sebenarnya versi iteratif tidak jauh berbeda dengan versi rekursif. Pemakaian kedua tipe itupun sangat bergantung pada keadaan. Ada keadaan dimana lebih cocok bila memakai versi rekursif, dan ada pula dimana kita harus memakai versi iteratif. Dalam makalah ini akan dibatasi hanya membahas versi iteratif saja.

Algoritma runut-balik banyak diterapkan pada bidang *games* pencarian solusi (misalnya catur, labirin, dsb) dan kecerdasan buatan (*artificial intelligence*). Dalam makalah ini akan dibahas penerapan algoritma runut-balik iteratif untuk mencari solusi pada permainan mengasah otak dari Jepang yaitu *sudoku* dan optimasi algoritma iteratif tersebut.

Kata kunci: *runut-balik, DFS, brute-force, sudoku*

1. PENDAHULUAN

Penerapan algoritma runut-balik untuk mencari solusi pada permainan *sudoku* sebenarnya sangat sederhana. Algoritma runut-balik versi iteratif ini sudah pernah diimplementasikan oleh penulis pada Tugas Besar II Strategi Algoritmik. Penulis menamakan algoritma ini dengan nama *brute-tracking*, karena merupakan gabungan dari algoritma *brute-force* dan runut-balik (*backtracking*). Penulis juga membuat suatu algoritma lagi untuk optimasi algoritma *brute-tracking* tersebut. Algoritma optimasi itu dinamakan *findtracking* (runut-cari), karena memang algoritma ini secara garis besar adalah runut-balik dengan pencarian. Dua algoritma ini berhasil untuk mencari solusi dalam permainan *sudoku*.

Penjelasan lebih rinci mengenai dua algoritma runut-balik iteratif pencarian solusi pada permainan *sudoku* ini yaitu *brute-tracking* dan *findtracking* akan dibahas di dalam bab metode.

2. METODE

Berikut penjelasan lebih rinci mengenai algoritma *brute-tracking* dan *findtracking*. Penulis membuat dua algoritma ini memakai bahasa Java dan telah lulus kompilasi di lingkungan Java Runtime Environment 6.0

2.1 Algoritma *Brute-Tracking*

Konsepnya sangat sederhana, yaitu pengecekan angka X (1<=X<=n, dengan n adalah besar sisi papan *sudoku*) ke dalam sebuah elemen matriks pada papan *sudoku* dimulai dari elemen matriks kosong pada baris dan kolom paling atas. Bila memenuhi syarat, yaitu tidak ada angka yang sama dengan X pada baris, kolom, dan *zone* yang sama, maka X akan dimasukkan ke dalam elemen matriks dan indeks kolom atau baris akan maju (tergantung format *programmer*) dan dilakukan pengecekan ulang lagi sampai elemen matriks kosong baris dan kolom paling bawah. Bila tidak ada memenuhi syarat, maka X akan bertambah satu dan dilakukan pengecekan lagi. Bila angka telah mencapai n tetapi tetap tidak memenuhi syarat, maka dilakukan *backtrack* dengan cara mundur satu kolom atau satu baris (tergantung format *programmer*) dan dilakukan penambahan satu pada nilai elemen matriks hasil mundur tersebut dan dilakukan kembali pengecekan ulang, dan seterusnya hingga dicapai diantara dua *state* yaitu ditemukannya solusi (semua elemen matriks terisi dan memenuhi syarat) atau tidak ditemukannya solusi (*backtrack* telah mencapai elemen matriks paling awal yang diisi dan elemen matriks itu adalah angka n namun tetap tidak memenuhi syarat). Dapat dilihat bahwa pengecekan angka dari 1-n adalah sifat dari algoritma

brute-force dan kemampuan mundur indeks baris atau kolom adalah sifat dari algoritma runut-balik.

Berikut adalah kode dari algoritma *brute-tracking* dalam bahasa Java

```
public void BruteTracking(MatriksSudoku T) {

  /* inisiasi */
  int i=1,j=1,temp=0,angka=1;

  /* getter ukuran sudoku */
  int max=T.getSize();

  /* cari baris terkecil dimana elemen
     matriksnya nol */
  int b=T.whoAwal(max,'b');

  /* cari kolom terkecil dimana elemen
     matriksnya nol */
  int k=T.whoAwal(max,'k');

  /* inisiasi awal syarat matriks masih
     berpeluang mempunyai solusi */
  T.setIsi(b,k,1,false);


  /* proses, terus looping hingga ditemukan
     solusi atau tidak ditemukan solusi */
  while ( (i!=(max+1)) && (T.getIsi(b,k)!=0) ) {
    /* cek status elemen matriks, true jika dari
       user atau dibangkitkan dari soal, false
       jika masih kosong */
    if (T.getStat(i,j)==false) {
      /* status elemen matriks adalah kosong,
         mulai percobaan mengisi nilai elemen */
      T.setIsi(i,j,angka,false);
      /* cek keadaan apakah nilai masukan ada di
         baris, kolom, atau zone yang sama */
      if (T.isSyarat(i,j,angka)==true) {
        /* syarat terpenuhi, isi elemen dengan
           nilai angka */
        T.setIsi(i,j,angka,false);
        /* cek indeks kolom */
        if (j==max) {
          /* indeks kolom ada di max, maka
             indeks baris maju dan kolom = 1*/
          j=1;
          i++;
        }
        else {
          /* indeks kolom<max, tambah indeks */
          j++;
        }
        /* reset inisiasi pengisian nilai */
        angka=1;
      }
      else {
        /* syarat tidak terpenuhi, cek apakah
           inisiasi pengisian nilai sama
           dengan max */
        if (angka==max) {
          /* angka = max, maka nilai elemen
             matriks di-reset jadi kosong */
          T.setIsi(i,j,0,false);
          /* memulai backtracking, mundurkan
             indeks baris atau kolom */

          do {
            /* cek apakah indeks kolom = 1 */
            if (j==1) {
              /* backtrack dengan memundurkan
                 indeks baris, dan indeks
                 kolom menjadi max */
              i--;
              j=max;
            }
            else {
              /* backtrack dengan memundurkan
                 indeks kolom */
              j--;
            }
            /* looping  hingga mendapatkan
               elemen matriks berstatus false */
          } while (T.getStat(i,j)==true);
          /* isi nilai elemen hasil backtrack ke
             variabel temporer */
          temp=T.getIsi(i,j);
          /* cek nilainya apakah temp = max */
          if (temp==max) {
            /* nilai = max, maka reset nilai
               elemen lagi */
            T.setIsi(i,j,0,false);
            /* memulai backtracking lagi, cara
               sama seperti diatas */
            do {
              if (j==1) {
                i--;
                j=max;
              }
              else {
                j--;
              }
            } while (T.getStat(i,j)==true);
            /* isi nilai elemen hasil backtrack
               ke variabel temporer lagi */
            temp=T.getIsi(i,j);
            /* cek lagi apakah temp = max
            if (temp==max) {
              /* nilai = max, maka reset nilai
                 elemen lagi */
              T.setisi(i,j,0,false);
              /* memulai backtracking lagi, cara
                 sama seperti diatas */
              do {
                if (j==1) {
                  i--;
                  j=max;
                }
                else {
                  j--;
                }
              } while (getstat(i,j)==true);
            }
            else {
              /* temp /= max, maka inisiasi
                 pengisian nilai menjadi
                 temp ditambah satu */
              angka=tmp+1;
            }
          }
          else {
            /* temp /= max, maka inisiasi
                 pengisian nilai menjadi
                 temp ditambah satu */
            angka=tmp+1;
          }
        }
        else {
          /* angka belum mencapai max, maka
```

```
            tambahkan satu */
          angka++;
        }
      }
    }
    else {
      /* elemen matriks adalah masukan user atau
         dibangkitka dari user, maka majukan
         indeks baris atau kolom */
      if (j==max) {
        /* indeks kolom = max, maka indeks
           baris maju dan kolom = 1 */
        j=1;
        i++;
      }
      else {
        /* indeks kolom<max, indeks kolom maju */
        j++;
      }
      /* reset inisiasi pengisian nilai */
      angka=1;
    }
    /* gambarkan perubahan, mencetak matriks */
    cetak();
  }
}
```

Walaupun kita lihat bahwa algoritma *brute-tracking* ini cukup mudah dalam pengerjaan dan pembuatan kodenya, tetapi tetap saja algoritma ini tidak begitu mangkus untuk pencarian solusi *sudoku*. Karena, algoritma ini berjalan secara searah dari kiri atas ke kanan bawah. Sehingga sebelum mengisi harus diperiksa terlebih dahulu apakah elemen matriks adalah masukan user, dibangkitkan dari soal, atau bukan keduanya. Juga, algoritma ini bergantung pada besarnya sisi papan *sudoku*. Pada kasus besar sisi 16 atau lebih, algoritma ini menjadi terkesan "bodoh" dan terlihat ketidakmangkusannya. Atas dasar itulah penulis berpikir untuk membuat suatu algoritma yang lebih "pintar" dan optimal daripada algoritma *brute-tracking* tersebut. Selanjutnya akan dibahas mengenai algoritma optimasi iteratif ini.

2.1 Algoritma *Findtracking* (Runut-cari)

Konsep algoritma ini tidak terlalu rumit. Sesuai namanya, algoritma ini bertujuan untuk mengoptimalkan pencarian solusi dengan cara mencari terlebih dahulu baris, kolom, atau *zone* yang telah terisi angka paling banyak tetapi bukan yang telah terisi semua. Setelah didapatkan baris, kolom, atau *zone* tersebut, lalu dicari lagi dari baris, kolom, atau *zone* tersebut elemen matriks yang masih kosong yang mempunyai baris dan kolom paling kecil. Setelah ditemukan elemen matriks tersebut, barulah dilakukan pengecekan dengan sebuah angka X (1<=X<=n, dengan n adalah besar sisi papan *sudoku*). Bila X memenuhi syarat (tidak ada angka X dalam baris, kolom dan *zone* yang sama) maka X akan dimasukkan ke dalam elemen matriks itu dan akan dilakukan pencarian optimasi lagi (yaitu mencari baris, kolom, atau *zone* yang memiliki elemen matriks telah terisi paling banyak, sama seperti sebelumnya). Setelah didapatkan, baris dan kolom elemen matriks sebelumnya disimpan di dalam atribut elemen matriks yang baru ditemukan tadi agar nanti dapat di-*backtrack*. Bila X tidak ada yang memenuhi syarat, maka akan di-*backtrack* ke baris dan kolom yang telah disimpan sebelumnya di dalam atribut elemen matriks tersebut. Setelah itu nilai elemen pada baris dan kolom yang disimpan tersebut akan ditambah satu dan dilakukan pengecekan ulang dan pencarian optimasi lagi hingga tercapai diantara dua *state* yaitu ditemukannya solusi (semua elemen matriks terisi dan memenuhi syarat) atau tidak ditemukannya solusi (*backtrack* telah mencapai elemen matriks paling awal yang diisi dan elemen matriks itu adalah angka n namun tetap tidak memenuhi syarat). Dapat dilihat disini optimasi dilakukan pada pencarian baris dan kolom setelah pengisian berhasil dan *backtrack* dengan cara penyimpanan indeks pengisian sebelumnya. Dengan optimasi seperti ini *backtracking* akan menjadi lebih "pintar" dan lebih mangkus dalam hal pengerjaan dan waktu dibandingkan dengan algoritma *brute-tracking*. Tetapi pada kasus matriks *sudoku* kosong, cara penyelesaiannya akan sama persis dengan algoritma *brute-tracking*, karena pencarian optimasi pada baris, kolom, atau *zone* akan selalu menghasilkan nilai kembalian nol dan akan diambil baris dan kolom terkecil dari elemen matriks kosong.

Berikut adalah kode optimasi dan algoritma runut-cari dalam bahasa Java

```
public Point Optimasi(MatriksSudoku T) {
  /* method tambahan untuk mencari indeks baris
     dan kolom optimal yang dikembalikan dalam
     bentuk point */
  /* cari indeks baris yang terisi terbanyak */
  int idxB = MaxBaris(T);
  /* cari indeks kolom yang terisi terbanyak */
  int idxK = MaxKolom(T);
  /* cari indeks zone yang terisi terbanyak */
  int idxZ = MaxZone(T);
  /* bandingkan ketiganya, cari yang  terbesar */
  int Maks = Max(idxB,idxK,idxZ);
  /* cari siapa yang terbesar, 1=baris, 2=kolom,
     3=zone */
  int who = WhoMax(idxB,idxK,idxZ);
  /* cari elemen matriks kosong pertama dari
     indeks Maks dan who tersebut */
  Point p = new Point();
  p = CariIdxKosong(Maks,who);
  return p;
}

public void RunutCari(MatriksSudoku T) {

  /* inisiasi */
  int i=1,j=1,temp=0,angka=1,x,y;
  Point P = new Point();

  /* getter ukuran sudoku */
  int max=T.getSize();

  /* optimasi awal, cari baris dan kolom dimana
     elemen matriksnya nol */
```

```
P=Optimasi(T);
b=P.getY();
k=P.getX();

/* inisiasi awal syarat matriks masih
   berpeluang mempunyai solusi */
T.setIsi(b,k,1,false);

/* proses, terus looping hingga ditemukan
   solusi atau tidak ditemukan solusi */
while ( (!T.isFull()) && (T.getIsi(b,k)!=0) ) {
  /* tidak perlu cek status elemen matriks,
     karena dari hasil optimasi pasti
     didapatkan indeks baris dan kolom elemen
     matriks yang kosong*/
  T.setIsi(i,j,angka,false);
  /* cek keadaan apakah nilai masukan ada di
     baris, kolom, atau zone yang sama */
  if (T.isSyarat(i,j,angka)==true) {
    /* syarat terpenuhi, isi elemen dengan
       nilai angka */
    T.setIsi(i,j,angka,false);
    /* cari indeks baris dan kolom optimal */
    P=Optimasi(T);
    x=P.getX();
    y=P.getY();
    /* simpan indeks baris dan kolom saat ini
       di atribut indeks optimal yang baru */
    T.setPrev(y,x,i,j);
    /* set baris dan kolom sekarang */
    i=y;
    j=x;
    /* reset inisiasi pengisian nilai */
    angka=1;
  }
  else {
    /* syarat tak terpenuhi, cek apakah
       inisiasi pengisian nilai sama
       dengan max */
    if (angka==max) {
      /* angka = max, maka nilai elemen
         matriks di-reset jadi kosong */
      T.setIsi(i,j,0,false);
      /* memulai backtracking, mundurkan
         indeks baris atau kolom dengan cara
         melihat previous indeks yang telah
         disimpan di dalam atribut */
      y=T.getPrevB(i,j);
      x=T.getPrevK(i,j);
      /* inisiasi menjadi indeks sekarang */
      i=y;
      j=x;
      /* isi nilai elemen hasil backtrack ke
         variabel temporer */
      temp=T.getIsi(i,j);
      /* cek nilainya apakah temp = max */
      if (temp==max) {
        /* nilai = max, maka reset nilai
           elemen lagi */
        T.setIsi(i,j,0,false);
        /* memulai backtracking lagi, cara
           sama seperti diatas */
        y=T.getPrevB(i,j);
        x=T.getPrevK(i,j);
        /* inisiasi menjadi indeks sekarang*/
        i=y;
        j=x;
        /* isi nilai elemen hasil backtrack
           ke variabel temporer lagi */
        temp=T.getIsi(i,j);
        /* cek lagi apakah temp = max
        if (temp==max) {
          /* nilai = max, maka reset nilai
             elemen lagi */
          T.setisi(i,j,0,false);
          /* memulai backtracking lagi, cara
             sama seperti diatas */
          y=T.getPrevB(i,j);
          x=T.getPrevK(i,j);
          /* inisiasi menjadi indeks
             sekarang */
          i=y;
          j=x;
        }
        else {
          /* temp /= max, maka inisiasi
             pengisian nilai menjadi
             temp ditambah satu */
          angka=tmp+1;
        }
      }
      else {
          /* temp /= max, maka inisiasi
               pengisian nilai menjadi
               temp ditambah satu */
          angka=tmp+1;
        }
      }
      else {
        /* angka belum mencapai max, maka
           tambahkan satu */
        angka++;
      }
    }
     /* gambarkan perubahan, mencetak matriks */
    cetak();
  }
}
```

Walaupun algoritma ini lebih mangkus daripada algoritma *brute-tracking* (karena tidak perlu pengecekan apakah masukan user, dibangkitkan soal, atau tidak), tetapi tetap saja algoritma ini masih belum dapat menjadi algoritma yang tepat untuk pencarian solusi permainan *sudoku*. Sebab, algoritma ini terlalu fokus pada pencarian, sehingga untuk kasus sisi *sudoku* yang besar pun akan memakan *resource* memori dan waktu yang besar pula karena optimasi pencariannya pun menjadi semakin sulit.

3. KESIMPULAN

- Algoritma *brute-tracking* dan *findtracking* dapat digunakan untuk pencarian solusi permainan *sudoku*
- Algoritma *findtracking* terbukti lebih mangkus daripada *brute-tracking* dalam hal pencarian solusi
- Algoritma *brute-tracking* dan *findtracking* belum dapat menjadi algoritma yang tepat untuk pencarian solusi permainan *sudoku*

REFERENSI

[1] Munir, Rinaldi. (2006). *Diktat Kuliah IF2251 Strategi Algoritmik*. Program Studi Teknik Informatika, Institut Teknologi Bandung