Tugas Kecil Strategi Algoritma

Laporan Implementasi Penyelesaian Persoalan 15-Puzzle dengan Algoritma *Branch and Bound*

Oleh:

David Karel Halomoan

13520154

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2022

A. Algoritma *Branch and Bound*

Algoritma *branch and bound* adalah algoritma yang menggabungkan mekanisme BFS (*Breadth First Search*) dengan teknik *leas cost search*. Algoritma ini digunakan untuk menyelesaikan persoalan optimasi, optimasi di sini adalah meminimalkan atau memaksimalkan suatu fungsi objektif, yang tidak melanggar batasan (*constraint*) persoalan. Perbedaan algoritma *branch and bound* dengan BFS adalah pada algoritma BFS murni, simpul diekspansi berdasarkan urutan pembangkitannya (FIFO, *First In First Out*), sedangkan pada algoritma *branch and bound*, setiap simpul diberi sebuah nilai *cost* (harga), nilai *cost* ini merupakan nilai taksiran lintasan terbaik (bisa termahal atau termurah, tergantung kasus persoalan) ke simpul status tujuan dari simpul nilai *cost* ini berada, simpul tidak lagi diekspansi berdasarkan urutan pembangkitannya, tetapi simpul yang memiliki *cost* yang paling optimal (*least cost search*) (bisa paling besar atau kecil, tergantung kasus persoalan).

Algoritma *branch and bound* juga menerapkan "pemangkasan" pada jalur yang dianggap tidak lagi mengarah pada solusi seperti pada algoritma *backtracking*. Algoritma akan memangkas simpul yang memiliki nilai tidak lebih baik dari nilai simpul terbaik sejauh ini dan simpul yang melanggar suatu batasan tertentu (simpul tidak merepresentasikan solusi yang *feasible*). Algoritma juga akan memangkas simpul jika solusi pada simpul tersebut hanya satu titik (tidak ada solusi lain), nilai fungsi objekif simpul ini akan dibandingkan dengan nilai fungsi objektif dari solusi terbaik saat ini, jika nilai fungsi objektif simpul ini lebih baik daripada nilai fungsi objektif solusi terbaik saat ini, simpul ini menjadi solusi terbaik.

Pada tugas kecil ini, penulis mendapatkan tugas untuk mengimplementasikan penyelesaian persoalan 15-Puzzle dengan algoritma *branch and bound*. Penulis memutuskan untuk melakukan implementasi dengan Bahasa pemrograman Java. Permainan 15-Puzzle adalah permainan yang dapat dimenangkan jika pemain berhasil mengubah konfigurasi ubin menjadi konfigurasi final (ubin terurut berdasarkan nomornya dan tidak terdapat ubin (kosong) pada posisi ujung bawah kanan dari *puzzle*. Konfigurasi ubin hanya dapat diubah dengan menggeser ubin yang langsung menyentuh posisi

kosong ke posisi kosong tersebut, posisi kosong yang baru pun akan berubah menjadi posisi lama ubin yang digeser tersebut.

1	3	4	15
2		5	12
7	6	11	14
8	9	10	13

(a) Susunan awal

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	

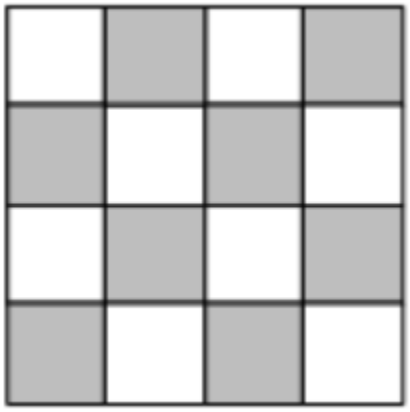
(b) Susunan akhir

Gambar 1 Contoh Konfigurasi Awal dan Konfigurasi Final

Selanjutnya, akan dijelaskan langkah-langkah pada algoritma yang penulis buat. Pertama-tama, program harus menentukan konfigurasi ubin yang dimasukkan dapat diselesaikan atau tidak. Terdapat $(16!)/2$ konfigurasi ubin berbeda yang dapat dicapai dari suatu konfigurasi wala sembarang. Program menentukan konfigurasi ubin dapat dieselesaikan dengan menggunakan rumus:

$$\Sigma_{i=1}^{16} KURANG(i) + X$$

Konfigurasi dapat diselesaikan jika nilai dari rumus tersebut genap. Nilai X pada rumus tersebut adalah 1 jika sel kosong berada pada posisi awal pada sel yang diarsir sesuai Gambar 2.



Gambar 2 Posisi arsir pada puzzle untuk menentukan nilai X

Nilai $KURANG(i)$ pada rumus tersebut adalah banyaknya ubin bernomor *j* sedemikian sehingga $j < i$ dan $POSISI(i) > POSISI(j)$, dengan $POSISI(i)$ adalah posisi ubin bernomor i pada konfigurasi yang diperiksa. Program melakukan pencarian nilai rumus ini dengan kompleksitas $O(n^2)$ (diasumsikan masukan berukuran n karena masukan selalu berukuran $4 \times 4 = 16$). Program juga memeriksa posisi kosong dengan menandai posisi kosong sebagai ubin bernomor bilangan bulat (*integer*) 16. Program akan menampilkan pesan yang menunjukkan konfigurasi masukan tidak dapat dieselesaikan jika nilai rumus tersebut ganjil. Jika nilai rumus tersebut genap, program akan melanjutkan ke tahap selanjutnya. Pada tahap selanjutnya, program akan menghitung mulai menghitung nilai *cost* dari simpul yang dibangkitkan, nilai *cost* akan dihitung dengan melakukan penjumlahan ongkos mencapai simpul i (diasumsikan simpul yang akan dibangkitkan adalah simpul i) dari akar dan ongkos mencapai simpul tujuan (konfigurasi final) dari simpul i. Ongkos mencapai simpul i dari akar adalah panjang lintasan dari simpul akar ke simpul i (kedalaman simpul dalam pohon), dengan ongkos dari akar sendiri adalah 0. Ongkos untuk mencapai simpul tujuan adalah taksiran panjang lintasan terpendek dari simpul i ke simpul tujuan pada upapohon berakar simpul i, taksiran ini didapatkan dengan menghitung jumlah ubin tidak kosong yang tidak berada pada posisi seharusnya (konfigurasi final). Program pertama memasukkan konfigurasi awal ke dalam sebuah *priority queue*. *Priority queue* pada program menggunakan *priority queue* yang terdapat pada kelas *Collections* pada Java. *Priority queue* ini diimplementasikan dengan *heap* sehingga hanya memiliki kompleksitas $O(log\ n)$ untuk memasukkan dan mengeluarkan data dari *priority queue*. *Priority queue* akan menyimpan elemen dengan memprioritaskan elemen dengan nilai *cost* terkecil (makin kecil nilai *cost*, makin dekat menuju *head* dari *priority queue*). Elemen yang disimpan dalam *priority queue* adalah instansiasi dari kelas State yang menyimpan konfigurasi ubin, koordinat posisi ubin kosong, nilai *cost* dari konfigurasi tersebut, kedalaman simpul tersebut, dan sebuah sebuah struktur data bertipe DirectionNode yang menyimpan blangan bulat yang melambangkan arah gerakan ubin kosong untuk mencapai konfigurasi ini dari konfigurasi sebelumnya (disebut gerakan terakhir) dan

DirectionNode dari simpul sebelumnya yang membangkitkan simpul ini (dilambangkan dengan atribut next), penggunaan DirectionNode digunakan untuk menjaga keefektifan penggunaan memori (simpul yang memiliki orang tua yang sama tidak membuat daftar langkah yang sama untuk setiap simpul tersebut, tetapi "menunjuk" ke daftar langkah yang sama yang ada pada simpul orangtuanya, DirectionNode ini mirip seperti struktur data *linked list*). Arah gerakan ke atas dilambangkan dengan bilangan 1, arah ke bawah dengan bilangan -1, arah ke kanan dengan bilangan 2, dan arah ke kiri dengan bilangan -2. Untuk simpul akar, program akan memasukkan State baru dengan nilai *cost* 0, kedalaman 0, dan gerakan terakhir 0. Program lalu akan melakukan *loop* selama *priority queue* belum kosong. Sebelum melakukan *loop*, program menginisialisasi nilai batas *cost* (*limitCost*) dengan nilai yang besar. Langkah pertama dalam *loop* tersebut adalah mengeluarkan elemen yang berada pada *head priority queue* (di-*dequeue* dari *priority queue*). Program lalu akan mengecek kecocokan konfigurasi ubin dari elemen tersebut dengan konfigurasi final. Pengecekan dilakukan dengan terlebih dahulu mengecek posisi ubin kosong yang terdapat pada State yang di-*dequeue* tersebut dan *cost* State tersebut sudah melewati *limitCost* atau tidak. Ini dilakukan untuk mengurangi kejadian program mengecek seluruh elemen dalam konfigurasi ubin (meningkatkan efisiensi program). Jika State melewati proses pengecekan tersebut, program baru akan mencocokkan seluruh elemen konfigurasi ubin dengan konfigurasi final. Jika konfigurasi sesuai, program akan menandai bahwa suatu solusi telah ditemukan dan mengganti nilai *limitCost* dengan nilai *cost* pada State tersebut. Program juga akan menyimpan DirectionNode dari State tersebut (program menyimpan langkah yang diperlukan untuk mencapai konfigurasi final) dan menghilangkan semua elemen pada *priority queue* yang memiliki *cost* lebih dari *limitCost* yang baru. Program juga akan menyimpan konfigurasi pada sebuah *Hash Map* dengan *key* konfigurasi tersebut dan *value* berupa *cost* dari State tersebut. Program lalu akan mencoba membangkitkan konfigurasi ubin baru dengan urutan atas, kanan, bawah, kiri (ini merupakan arah Gerakan ubin kosong). Program tidak akan membangkitkan konfigurasi jika konfigurasi ubin akan kembali ke konfigurasi sebelumnya, program akan mengecek gerakan terakhir dari State dan

mencegah pembangkitan konfigurasi dengan arah berkebalikan dari gerakan terakhir tersebut (ini merupakan salah satu kegunaan dari DirectionNode). Program juga akan mencegah pembangkitan konfigurasi apabila ubin kosong berada di pinggir *puzzle* dan pergerakan akan mebuat ubin kosong "keluar" dari *puzzle*. Pada saat program mencoba membangkitkan suatu konfigurasi baru, program akan melakukan pengecekan *cost* konfigurasi tersebut lebih kecil atau kurang dari *limitCost* atau tidak. Jika tidak, program tidak akan membangkitkan konfigurasi tersebut. Jika iya, program akan mengecek konfigurasi ada pada *hash map* atau tidak, jika tidak, program akan mengecek *cost* dari konfigurasi yang akan dibangkitkan lebih kecil atau sama dengan nilai *value* pada *hash map* atau tidak. Jika tidak, program tidak akan membangkitkan konfigurasi tersebut. Jika iya, program akan membangkitkan State tersebut berdasarkan arah pembangkitan. Program akan memasukkan DirectionNode dari konfigurasi sebelumnya (simpul orang tua) sebagai elemen selanjutnya dari DirectionNode pada State tersebut. Program juga akan meng-*update* nilai *value* pada *hash map* dengan *key* konfigurasi tersebut dan memasukkan State ke *priority queue*. Jika elemen pada *priority queue* sudah habis dan program menemukan suatu solusi, program akan menampilkan jumlah langkah yang diperlukan dan urutan langkah-langkah tersebut dengan melakukan traversal pada DirectionNode solusi yang telah disimpan dan memasukkan tiap langkah ke dalam sebuah *stack* (ini dilakukan karena traversal menelusuri langkah dari akhir sampai awal). Program lalu akan menampilkan langkah-langkah dengan melakukan *pop* pada *loop* sampai *stack* tersebut kosong. Jika program tidak menemukan solusi, program akan menampilkan pesan bahwa solusi tidak ditemukan.

B. *Source Code* Program Dalam Bahasa Java

1. **Kelas State**

```java
import java.util.List;

public class State {
    public List<List<Integer>> puzzle;
    public int cost;
    public int depth;
    public int[] idxKosong = new int[2];
    public DirectionNode directionNode;

    public State(List<List<Integer>> puzzle, int cost, int depth, int row, int col, int lastMove) {
        this.puzzle = puzzle;
        this.cost = cost;
        this.depth = depth;
        this.idxKosong[0] = row;
        this.idxKosong[1] = col;
        this.directionNode = new DirectionNode();
        this.directionNode.direction = lastMove;
    }
}
```

2. **Kelas StateComparator (digunakan untu menentukan prioritas State dalam *priority queue*, mengimplementasikan *interface* Comparator)**

```java
import java.util.Comparator;

public class StateComparator implements Comparator<State> {
    public int compare(State s1, State s2) {
        if (s1.cost > s2.cost) {
            return 1;
        } else if (s1.cost < s2.cost) {
            return -1;
        }
        return 0;
    }
}
```

3. **Kelas DirectionNode**

```java
public class DirectionNode {
    public int direction;
    public DirectionNode next;
}
```

4. **Atribut Kelas BranchAndBound (Kelas yang digunakan untuk melakukan algoritma *branch and bound*)**

```java
public class BranchAndBound {
    // Atribut statik
    static private PriorityQueue<State> pq = new PriorityQueue<State>(new StateComparator());
    static private HashMap<List<List<Integer>>, Integer> myMap = new HashMap<List<List<Integer>>, Integer>();
    static private int limitCost = 999;
    static private List<List<Integer>> puzzle;
    static public Stack<Integer> st = new Stack<Integer>();
    static public int countNode = 0;
    static public float time;
    static public int stepCount;
    static public String[][] directionString;
    static private List<List<Integer>> goal = new ArrayList<List<Integer>>() {{
        add(List.of(1, 2, 3, 4));
        add(List.of(5, 6, 7, 8));
        add(List.of(9, 10, 11, 12));
        add(List.of(13, 14, 15, 16));
    }};
```

5. **Fungsi initiateNewPuzzle pada kelas BranchAndBound (digunakan untuk menginisialisasi *puzzle* baru yang akan di-*solve*)**

```java
static public void initiateNewPuzzle(List<List<Integer>> puzzle) {
        BranchAndBound.pq.clear();
        BranchAndBound.myMap.clear();
        BranchAndBound.st.clear();
        BranchAndBound.countNode = 1;
        BranchAndBound.limitCost = 999;
        int[] emptySlot = BranchAndBound.findEmptySlot(puzzle);
        BranchAndBound.puzzle = puzzle;
        BranchAndBound.pq.add(new State(puzzle, 0, 0, emptySlot[0], emptySlot[1], 0));

    }
```

6. **Fungsi findEmptySlot pada kelas BranchAndBound (digunakan untuk mencari posisi ubin kosong pada *puzzle*)**

```java
static private int[] findEmptySlot(List<List<Integer>> puzzle) {
        int[] ret = new int[2];
        boolean found = false;

        for (int i = 0; i < 4 && !found; i++) {
            for (int j = 0; j < 4 && !found; j++) {
                if (puzzle.get(i).get(j) == 16) {
                    found = true;
                    ret[0] = i;
                    ret[1] = j;
                }
            }
        }

        return ret;
    }
```

7. **Fungsi isSolveable pada kelas BranchAndBound (digunakan mengetahui *puzzle* dapat diselesaikan atau tidak)**

```
static private boolean isSolveable() {
        int sumKurang = 0, X = 0;
        for (int i = 0; i < 16; i++) {
            int kurang = 0;
            for (int j = i + 1; j < 16; j++) {
                if (puzzle.get(j / 4).get(j % 4) < puzzle.get(i / 4).get(i % 4)) {
                    kurang++;
                }
            }

            if (puzzle.get(i / 4).get(i % 4) == 16 && (((i / 4) % 2 == 0 && (i % 4) % 2 != 0) || ((i / 4) % 2 != 0 && (i % 4) % 2 == 0))) {
                X = 1;
            }

            sumKurang += kurang;
        }

        return (sumKurang + X) % 2 == 0;
    }
```

8. **Fungsi getCost pada kelas BranchAndBound (digunakan untuk mengetahui nilai *cost* dari suatu konfigurasi)**

```
static private int getCost(List<List<Integer>> mat) {
        int ret = 0;
        for (int i = 0; i < 4; i++) {
            for (int j = 0; j < 4; j++) {
                if (mat.get(i).get(j) != 16 && mat.get(i).get(j) != 4 * i + j + 1) {
                    ret++;
                }
            }
        }
        return ret;
    }
```

9. **Fungsi solve pada kelas BranchAndBound (digunakan untuk menyelesaikan *puzzle* dengan algoritma *branch and bound*)**

```
static boolean solve() {
        Instant start = Instant.now();
        if (!isSolveable()) {
            System.out.println("Tidak bisa diselesaikan!");
            return false;
        } else {
            DirectionNode d = null;
            boolean success = false;
            State s;
            while (!pq.isEmpty()) {
                s = pq.poll();

                if (s.idxKosong[0] == 3 && s.idxKosong[1] == 3 && s.cost < limitCost && s.puzzle.equals(goal)) {
                    int tempCost = s.cost;
                    success = true;
                    limitCost = tempCost;
                    d = s.directionNode;
                    BranchAndBound.stepCount = s.depth;
                    pq.removeIf(element -> element.cost > tempCost);
                }

                if (s.directionNode.direction != -1) {
                    up(s);
                }

                if (s.directionNode.direction != -2) {
                    right(s);
                }

                if (s.directionNode.direction != 1) {
                    down(s);
                }

                if (s.directionNode.direction != 2) {
                    left(s);
                }

            }
            BranchAndBound.time = Duration.between(start, Instant.now()).toMillis() / 1000.0F;

            if (success) {

                BranchAndBound.directionString = new String[BranchAndBound.stepCount][1];
                int i = BranchAndBound.stepCount - 1;

                while (d.direction != 0) {
                    st.push(d.direction);

                    if (d.direction == 1) {
                        BranchAndBound.directionString[i][0] = "UP";
                    } else if (d.direction == -1) {
                        BranchAndBound.directionString[i][0] = "DOWN";
                    } else if (d.direction == 2) {
                        BranchAndBound.directionString[i][0] = "RIGHT";
                    } else if (d.direction == -2) {
                        BranchAndBound.directionString[i][0] = "LEFT";
                    }

                    i--;
                    d = d.next;
                }

                System.out.println("Waktu yang dibutuhkan: " + BranchAndBound.time / 1000.0F + " s");
                System.out.println("Jumlah langkah: " + BranchAndBound.stepCount);
                System.out.println("Jumlah simpul yang dibangkitkan: "  + BranchAndBound.countNode);
                System.out.println("Urutan langkah: ");
                for (String step[]: BranchAndBound.directionString) {
                    System.out.println(step[0]);
                }
                return true;
            } else {
                System.out.println("Tidak bisa diselesaikan!");
                return false;
            }
        }
    }
```

10. Prosedur up, down, right, dan left (digunakan untuk mencoba membangkitkan simpul dengan ubin kosong bergerak sesuai arah yang ditentukan dari konfigurasi awal)

```java
static private void up(State s) {
      if (s.idxKosong[0] > 0) {
          List<List<Integer>> l = new ArrayList<List<Integer>>(4);

          for (List<Integer> el: s.puzzle) {
              l.add(new ArrayList<>(el));
          }

          int temp = l.get(s.idxKosong[0] - 1).get(s.idxKosong[1]);
          l.get(s.idxKosong[0] - 1).set(s.idxKosong[1], 16);
          l.get(s.idxKosong[0]).set(s.idxKosong[1], temp);

          int tempCost = getCost(l) + s.depth + 1;
          if (tempCost <= limitCost && (!myMap.containsKey(l) || tempCost <= myMap.get(l))) {
              State ret = new State(l, tempCost, s.depth + 1, s.idxKosong[0] - 1, s.idxKosong[1], 1);
              ret.directionNode.next = s.directionNode;
              myMap.put(l, tempCost);
              pq.add(ret);
              BranchAndBound.countNode++;
          }
      }
  }

  static private void down(State s) {
      if (s.idxKosong[0] < 3) {
          List<List<Integer>> l = new ArrayList<List<Integer>>(4);

          for (List<Integer> el: s.puzzle) {
              l.add(new ArrayList<>(el));
          }

          int temp = l.get(s.idxKosong[0] + 1).get(s.idxKosong[1]);
          l.get(s.idxKosong[0] + 1).set(s.idxKosong[1], 16);
          l.get(s.idxKosong[0]).set(s.idxKosong[1], temp);

          int tempCost = getCost(l) + s.depth + 1;
          if (tempCost <= limitCost && (!myMap.containsKey(l) || tempCost <= myMap.get(l))) {
              State ret = new State(l, tempCost, s.depth + 1, s.idxKosong[0] + 1, s.idxKosong[1], -1);
              ret.directionNode.next = s.directionNode;
              myMap.put(l, tempCost);
              pq.add(ret);
              BranchAndBound.countNode++;
          }
      }
  }

  static private void left(State s) {
      if (s.idxKosong[1] > 0) {
          List<List<Integer>> l = new ArrayList<List<Integer>>(4);

          for (List<Integer> el: s.puzzle) {
              l.add(new ArrayList<>(el));
          }

          int temp = l.get(s.idxKosong[0]).get(s.idxKosong[1] - 1);
          l.get(s.idxKosong[0]).set(s.idxKosong[1] - 1, 16);
          l.get(s.idxKosong[0]).set(s.idxKosong[1], temp);

          int tempCost = getCost(l) + s.depth + 1;
          if (tempCost <= limitCost && (!myMap.containsKey(l) || tempCost <= myMap.get(l))) {
              State ret = new State(l, tempCost, s.depth + 1, s.idxKosong[0], s.idxKosong[1] - 1, -2);
              ret.directionNode.next = s.directionNode;
              myMap.put(l, tempCost);
              pq.add(ret);
              BranchAndBound.countNode++;
          }
      }
  }

  static private void right(State s) {
      if (s.idxKosong[1] < 3) {
          List<List<Integer>> l = new ArrayList<List<Integer>>(4);

          for (List<Integer> el: s.puzzle) {
              l.add(new ArrayList<>(el));
          }

          int temp = l.get(s.idxKosong[0]).get(s.idxKosong[1] + 1);
          l.get(s.idxKosong[0]).set(s.idxKosong[1] + 1, 16);
          l.get(s.idxKosong[0]).set(s.idxKosong[1], temp);

          int tempCost = getCost(l) + s.depth + 1;
          if (tempCost <= limitCost && (!myMap.containsKey(l) || tempCost <= myMap.get(l))) {
              State ret = new State(l, tempCost, s.depth + 1, s.idxKosong[0], s.idxKosong[1] + 1, 2);
              ret.directionNode.next = s.directionNode;
              myMap.put(l, tempCost);
              pq.add(ret);
              BranchAndBound.countNode++;
          }
      }
  }
```

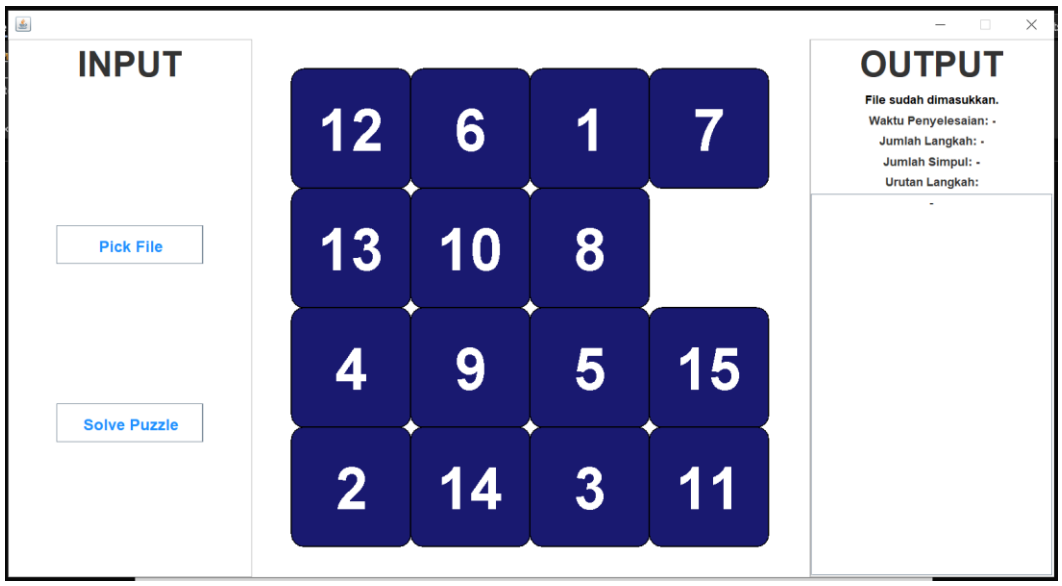
C. *Screenshot* dari *Input* dan *Output* Program

1. **Input notSolveable1.txt**

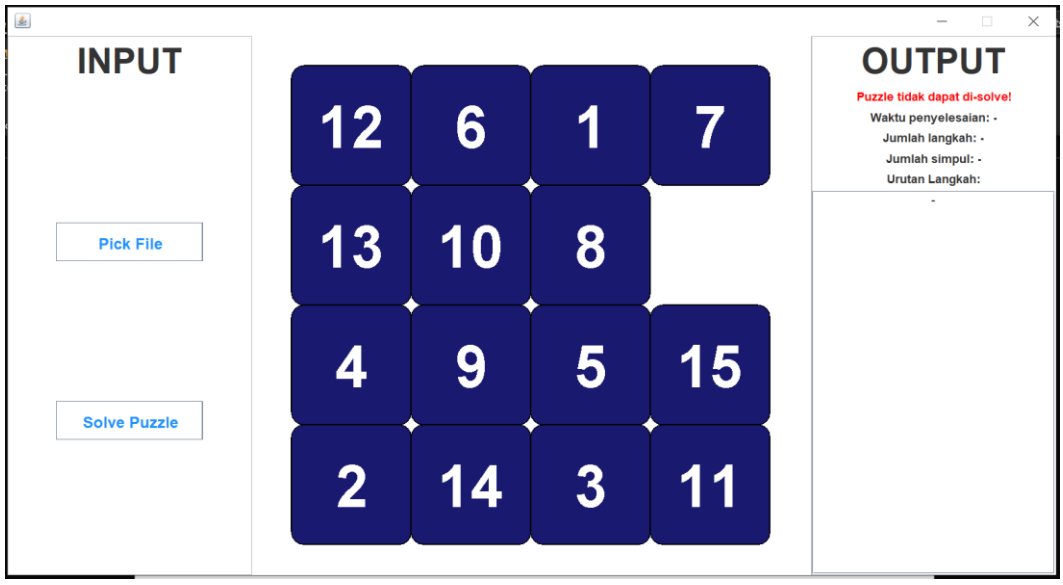
```
12 6 1 7
13 10 8 -
4 9 5 15
2 14 3 11
```

Output:

- **Awal:**



- **Akhir:**

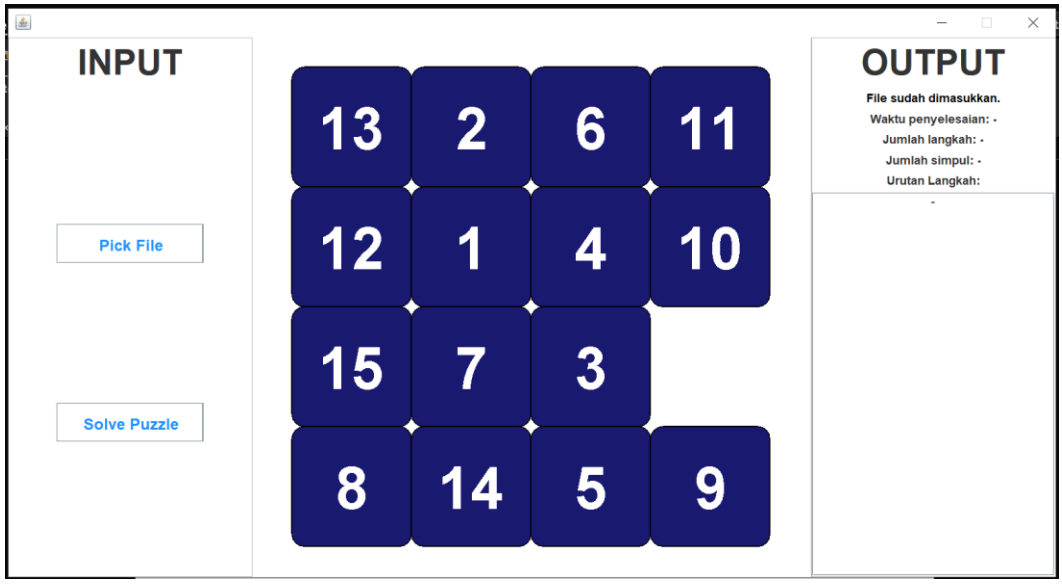


2. **Input notSolveable2.txt**

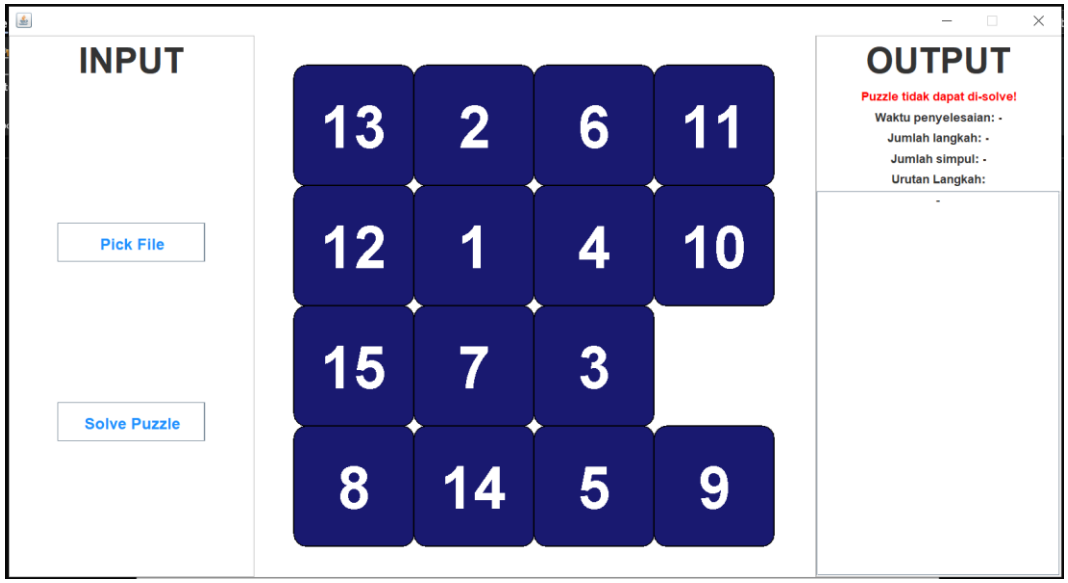
```
13 2 6 11
12 1 4 10
15 7 3 -
8 14 5 9
```

Output:

- **Awal:**



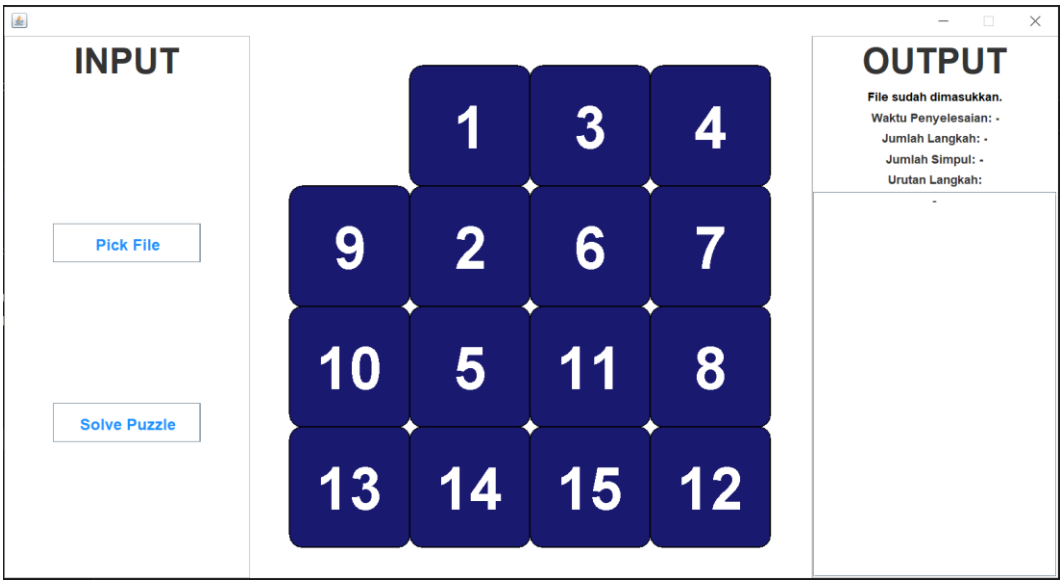
- **Akhir:**



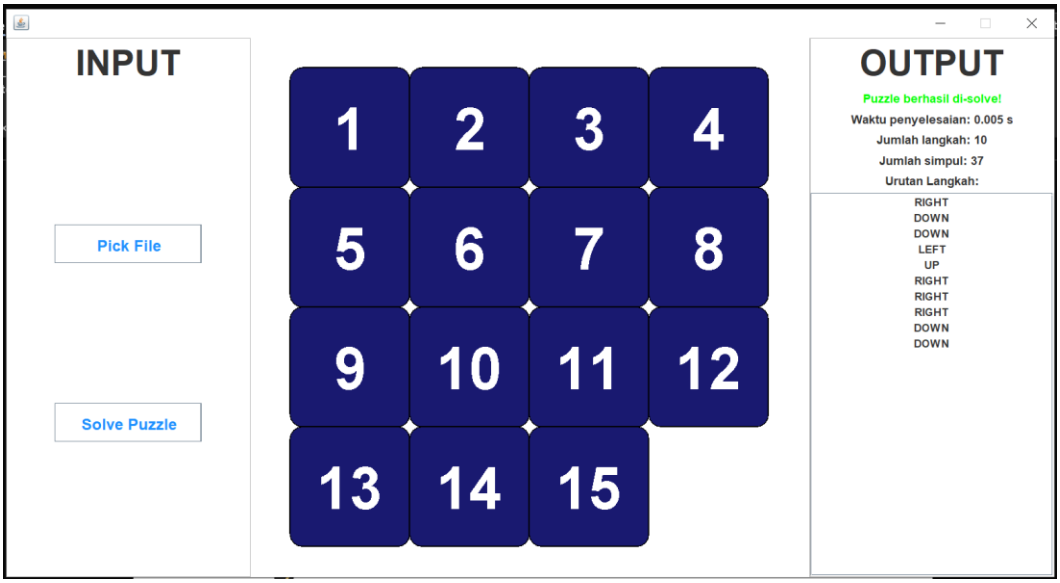
3. **Input solveable1.txt**

```
- 1 3 4
9 2 6 7
10 5 11 8
13 14 15 12
```

- **Awal:**



- **Akhir:**

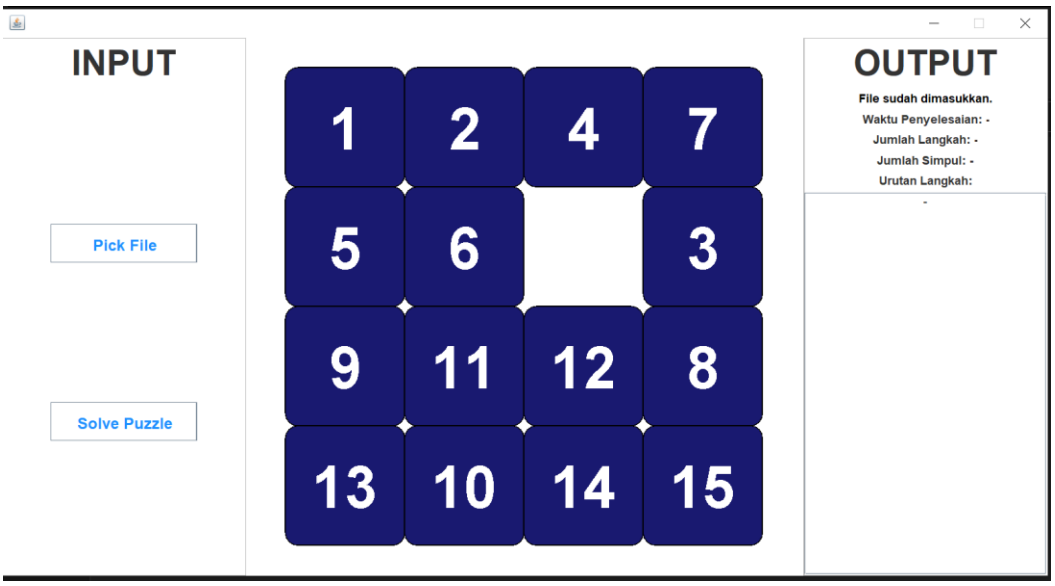


4. **Input solveable2.txt**

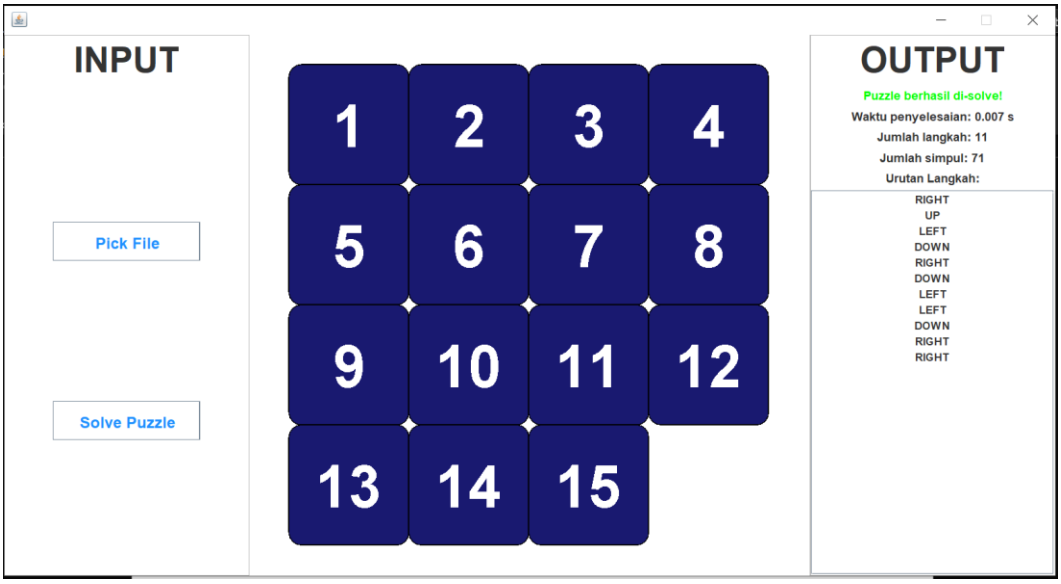
```
1 2 4 7
5 6 - 3
9 11 12 8
13 10 14 15
```

Output:

- **Awal:**



- **Akhir:**

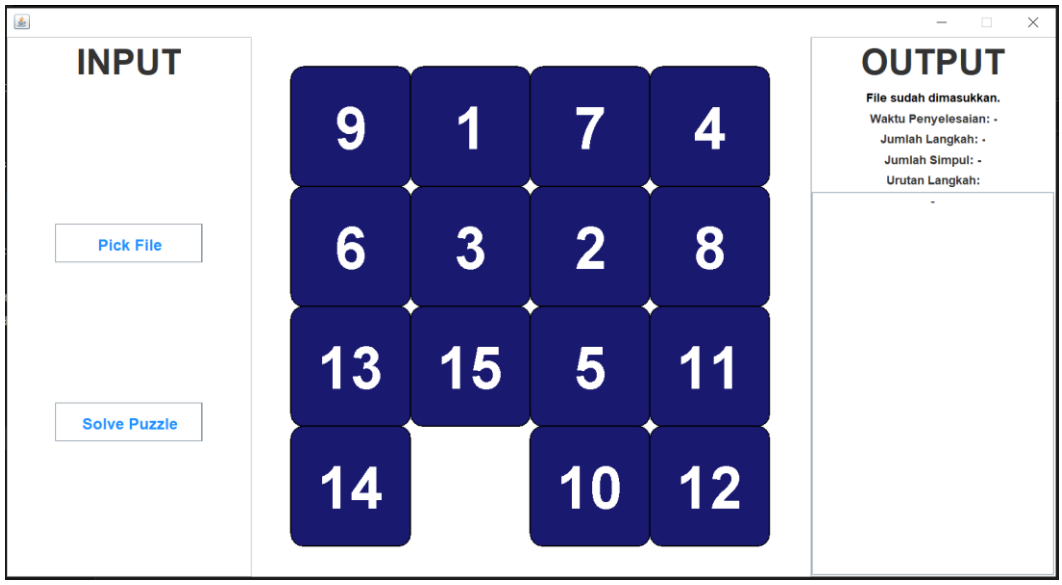


5. **Input solveable3.txt**

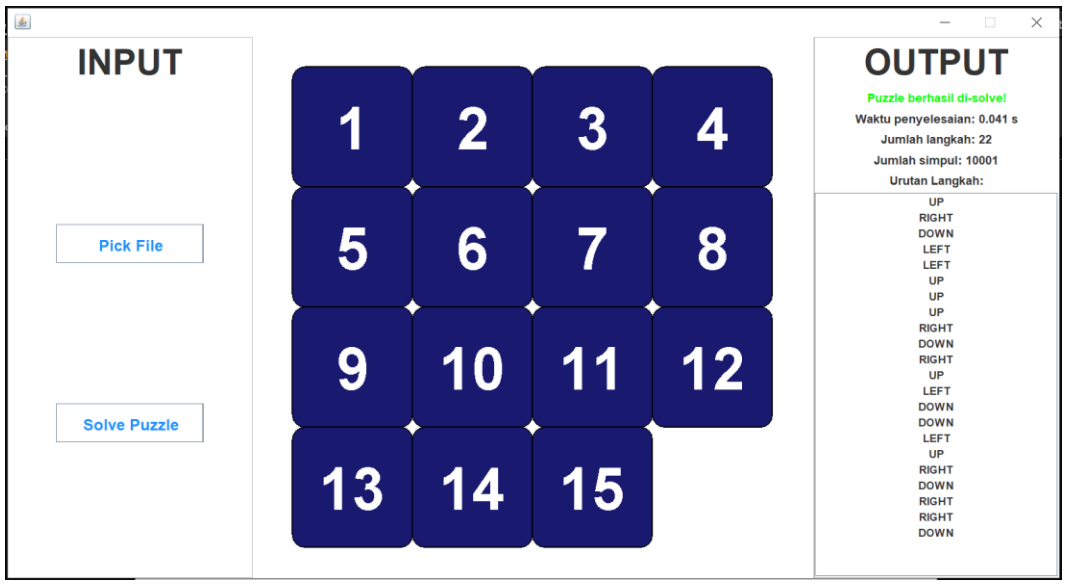
```
9 1 7 4
6 3 2 8
13 15 5 11
14 - 10 12
```

Output:

- **Awal:**



- **Akhir:**



D. Alamat Drive dan *Repository*

a. **Google Drive**
   https://drive.google.com/drive/folders/1OBDXnMyQaYgBy_bCz1rPn__rl_NFe_tW?usp=sharing
b. **Github *Repository***
   https://github.com/davidkarelh/Tucil3_13520154

E. Tabel Ceklist

Poin	**Ya**	**Tidak**
Program berhasil dikompilasi	√	
Program berhasil *running*	√	
Program dapat menerima input dan menuliskan output	√	
Luaran sudah benar untuk semua data uji	√	
Bonus dibuat	√	