BAB IV
IMPLEMENTASI DAN PENGUJIAN SISTEM

IV.1 Implementasi

Implementasi merupakan terjemahan perancangan yang berdasarkan hasil analisis pada bab sebelumnya ke dalam bahasa pemprograman yang dapat dimengerti oleh komputer. Bahasa pemprograman yang digunakan adalah *java*.

IV.1.1 Perangkat Lunak Implementasi dan Pengujian

Perangkat lunak yang digunakan untuk mengimplementasikan dan menguji aplikasi Rubik's Cube dengan BCO adalah sebagai berikut :

1. Windows 7 Ultimate sebagai sistem operasi
2. Netbeans 7.0 sebagai perangkat lunak pembangunan

IV.1.2 Perangkat Keras Implementasi dan Pengujian

Perangkat keras yang digunakan untuk mengimplementasikan dan menguji aplikasi Rubik's Cube dengan BCO adalah sebagai berikut:

1. Prosesor Intel Core i3-2350M 2.3GHz
2. RAM 2GB
3. Harddisk 250GB
4. Monitor 14"
5. Keyboard dan Mouse

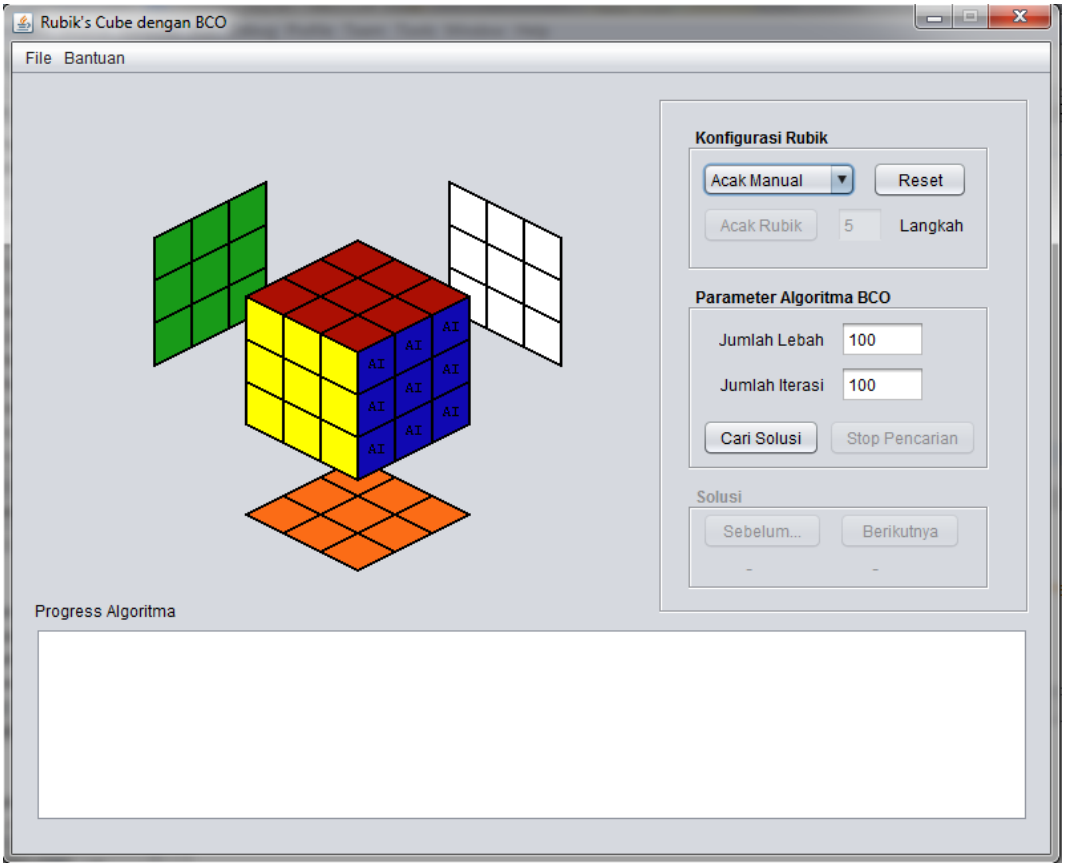
IV.1.3 Implementasi Antarmuka

Untuk memperjelas bentuk dari implementasi antarmuka berikut tampilan program yang telah dibuat :

1. Tampilan antarmuka *Form* Utama

   Tampilan antarmuka *Form* Utama memperlihatkan tampilan awal saat aplikasi dimulai yang terdiri dari simulasi Rubik's Cube, isian Jumlah Langkah untuk memasukkan jumlah langkah acak, tombol Acak Rubik untuk mengacak konfigurasi Rubik, tombol Reset Rubik untuk mengembalikan konfigurasi Rubik ke keadaan awal, isian Jumlah Lebah dan Jumlah Iterasi untuk memasukkan parameter jumlah lebah dan jumlah iterasi, tombol Cari Solusi untuk menjalankan algoritma pencarian, tombol Stop Pencarian untuk menghentikan pencarian, tombol Berikutnya dan Sebelumnya untuk menampilkan urutan langkah solusi serta kotak log Progress Algoritma untuk menampilkan progress algoritma yang sedang berjalan.

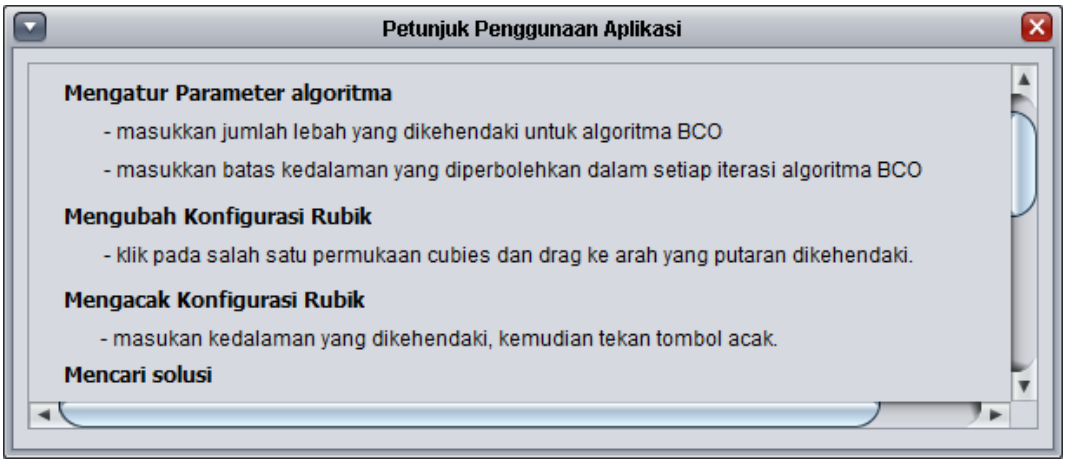
   Tampilan antarmuka *Form* Utama diperlihatkan pada Gambar IV.1.



Gambar IV.1 Tampilan *Form* Utama

2. Tampilan antarmuka *Form* Petunjuk

Tampilan jendela Petunjuk yang berisi petunjuk penggunaan aplikasi diperlihatkan pada Gambar IV.2.



Gambar IV.2 Tampilan *Form* Petunjuk

3. Tampilan antarmuka *Form* Tentang Pembuat

Tampilan jendela Tentang Pembuat yang berisi informasi tentang pembuat aplikasi diperlihatkan pada Gambar IV.3.



Gambar IV.3 Tampilan *Form* Tentang Pembuat

IV.2 Pengujian Perangkat Lunak

Pengujian perangkat lunak merupakan hal terpenting yang dilakukan untuk menemukan kekurangan atau kesalahan pada perangkat lunak yang diuji. Pengujian bermaksud untuk mengetahui perangkat lunak yang dibuat telah memenuhi kriteria yang sesuai dengan tujuan perancangan perangkat lunak. Adapun metode pengujian yang digunakan pada perangkat lunak ini adalah metode pengujian *black box dan* white box. Pengujian *black box* berfokus pada persyaratan fungsional perangkat lunak, sedangkan *white box* pengujian dengan melihat ke dalam modul untuk meneliti kode-kode program yang ada, dan menganalisis apakah ada kesalahan atau tidak.

IV.2.1 Pengujian White Box

Pengujian *white box* merupakan metode perancangan *test case* yang menggunakan struktur control dari perancangan metode dalam mendapatkan *test case*. Adapun metode yang digunakan dalam pengujian *white box* ini adalah metode *basis path.* Metode *basis path* mengijinkan pendesain kasus uji untuk mendapatkan perkiraan lojik yang kompleks dari desain metode dan menggunakan perkiraan ini untuk mendefenisikan aliran eksekusi.

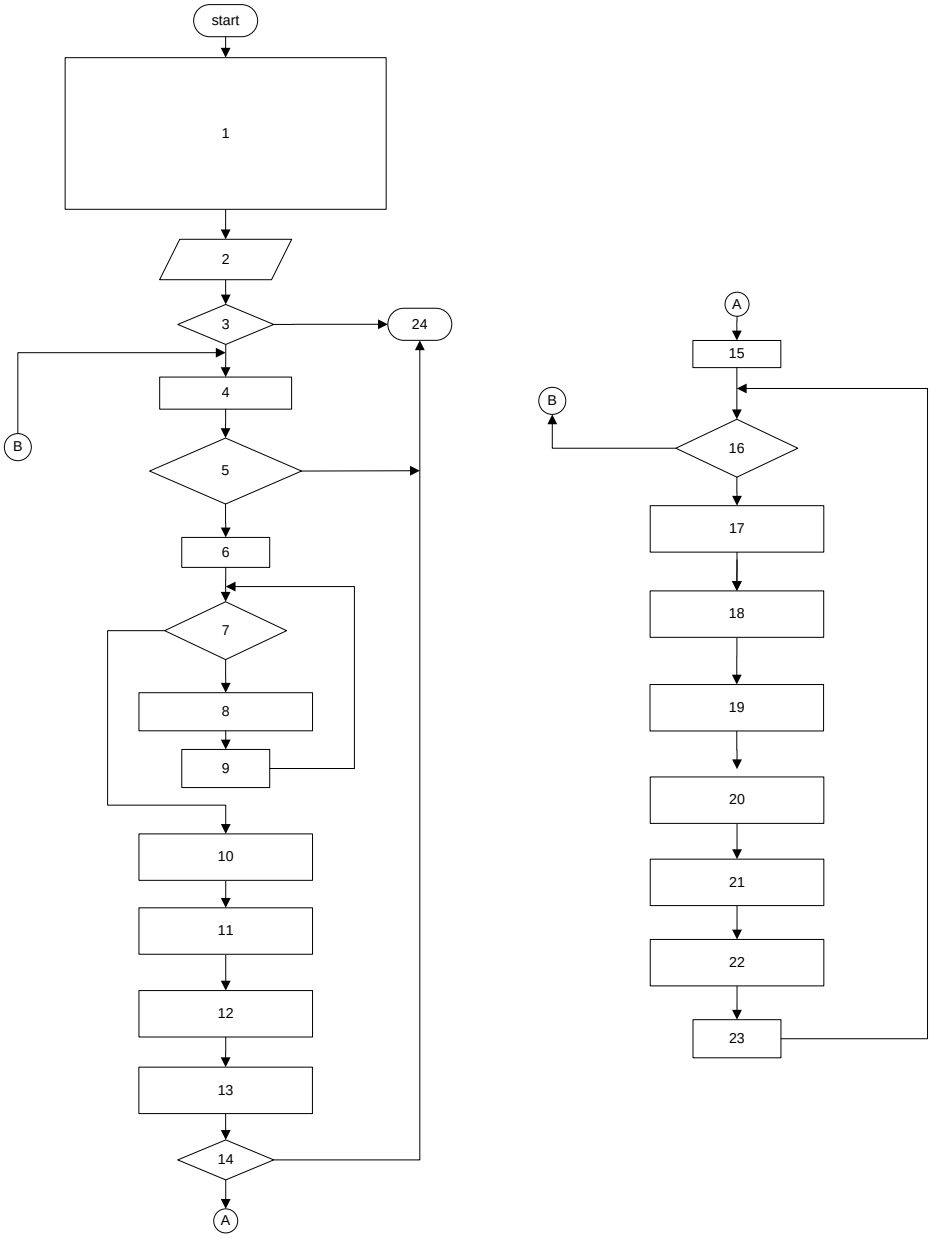
Pengujian *white box* akan digunakan pada algoritma *BCO*, untuk mengukur kinerja logika berdasarkan *flowchart* yang telah dibuat pada tahap analisis. Pengujian white box meliputi beberapa tahap yaitu :

1. Mengubah *source code* menjadi *flowchart.* Pengubahan *source code method* run yang berisi Algoritma BCO untuk mencari solusi Rubik’s Cube pada Gambar IV.4.

```
1   private final static String[] move = new String[13];
    private Cube cube;
    private Lebah[] koloni;
    private Node[] suksesor;
    private int jumLebah, maxDepth, jumIterasi, iterasi, jumNode;
    private Node startNode;
    public boolean stop;
    private long startTime;
    ArrayList<Lebah> recruiter;
    ArrayList<Lebah> follower;
    ArrayList<Lebah> distRecruiters;
    HashMap<Cube, ArrayList<Node>> grupDistSuksesor;
    public void run() {
2       stop = false;
        boolean ketemu = false;
        iterasi = 0;
3       while (!stop) {
4           koloni = new Lebah[jumLebah];
            suksesor = new Node[12];
            iterasi = iterasi + 1;
            int loop = 0;
5           if (iterasi > jumIterasi) {
              return;}
6           jumNode = 0;
            startNode = new Node();
            startNode.parent = null;
            startNode.cube = cube;
            startNode.depth = 0;
7           for (int i = 0; i < jumLebah; i++) {
8             koloni[i] = new Lebah(i, startNode);
              koloni[i].no = i;
9             koloni[i].recruiter = true; }
10          bangkitkanSuksesor(startNode, suksesor);
11          hitungfSuksesor(startNode, suksesor);
12          rouletteWheel1(suksesor, koloni);
13          cekKetemu(koloni);
14          while (!stop) {
15            loop = loop + 1;
16            if (loop == 41) {
                break; }
17            cekKetemu(koloni);
18            seleksiRecruiterFolower(recruiter, follower, koloni);
19            rouletteFollowerkeRecruiter(recruiter, follower);
20            grupRecruiter(recruiter, distRecruiters);
21            bangkitkanSuksesorDist(distRecruiters,  grupDistSuksesor);
22            rouletteFPkeN(distRecruiters, grupDistSuksesor);
23          }
24      }
```

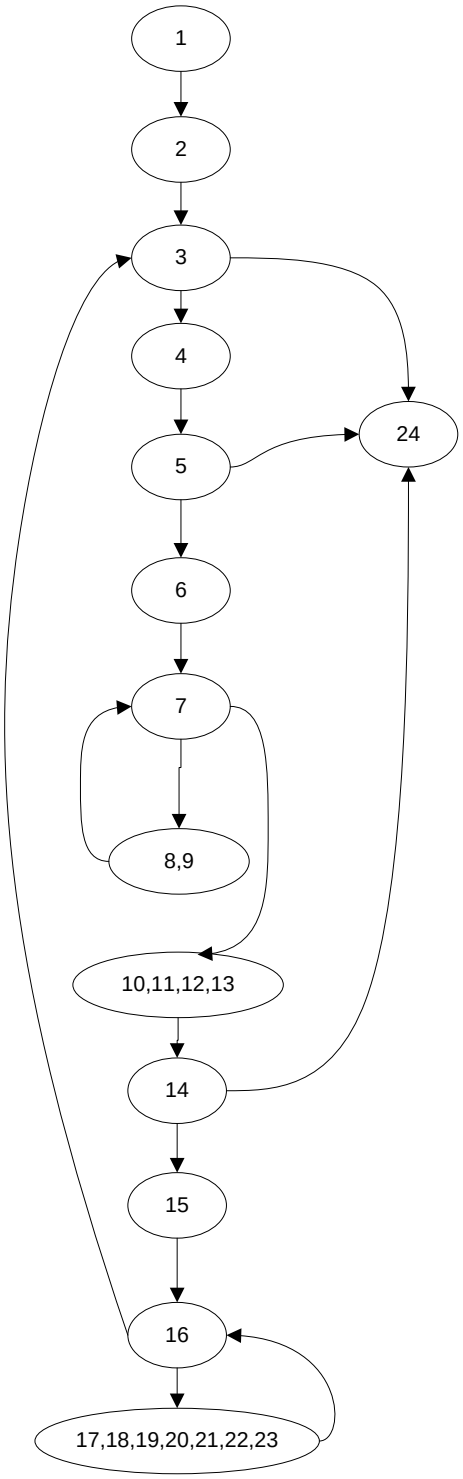
Gambar IV.4 *Source Code Method* run

Hasil pengubahan dan penomoran ke *flowchart* diperlihatkan pada Gambar IV.5.



Gambar IV.5 Penomoran *Flowchart* Algoritma BCO untuk puzzle Rubik's Cube

2. Mengubah *flowchart* menjadi *flowgraph*, diperlihatkan pada Gambar IV.6



Gambar IV.6 *Flowgraph* Algoritma BCO untuk *puzzle* Rubik's Cube

3. *Cyclomatic complexity*

   Dari diagram alir pada Gambar IV.6, dapat dihitung *cyclomatic complexity*, yakni :

   V(G) = 18 Busur – 14 Simpul + 2

   V(G) = 6

4. *Predicate node*

   V(G) = 5 *Predicate Node* + 1

   V(G) = 6

5. *Independent path*

   Dari hasil perhitungan *cyclomatic complexity* terdapat 6 *independent path* yaitu :

   Path 1: 1-2-3-24

   Path 2: 1-2-3-4-5-24

   Path 3: 1-2-3-4-5-6-7-10-11-12-13-14-24

   Path 4: 1-2-3-4-5-6-7-8-9-10-11-12-23-14-24

   Path 5: 1-2-3-4-5-6-7-10-11-12-13-14-15-16-3-24

   Path 6: 1-2-3-4-5-5-7-10-11-12-13-14-15-16-17-18-19-20-21-22-23-16-3-24

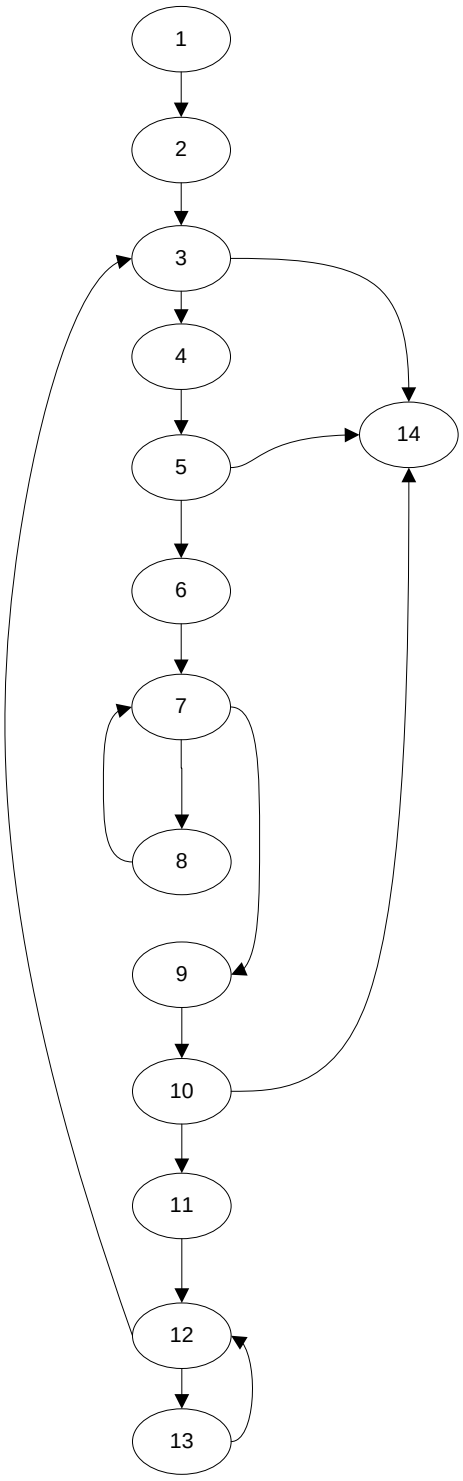
6. Hitung jumlah region

   Jumlah region = jumlah kurva tertutup + 1 (Region terluar)

   Jumlah region = 5 + 1

   Jumlah region = 6

7. Penomoran ulang *flowgraph*, diperlihatkan pada Gambar IV.7



Gambar IV.7 Penomoran ulang *Flowgraph*

8. *Graph Matrix*, diperlihatkan pada Tabel IV.1

Tabel IV.1 *Graph Matrix*

Node	1	2	3	4	5	6	7	8	9	10	11	12	13	14	Perhitungan
1		1													0
2			1												0
3				1										1	1
4					1										0
5						1								1	1
6							1								0
7								1	1						1
8							1								0
9										1					0
10											1			1	1
11												1			0
12			1										1		1
13												1			0
14															0
jumlah (x)															5

V(G) = x + 1

V(G) = 5 + 1

V(G) = 6

Berdasarkan pengujian pada setiap metode, dihasilkan nilai *Cyclomatic Complexity* yang sama yaitu 6. Maka dapat disimpulkan bahwa pengujian *white box* pada algoritma BCO berjalan dengan baik, karena setiap pengujian menghasilkan nilai yang sama.

IV.2.2 Pengujian *Black Box*

Pengujian *black box* adalah pengujian yang menyatakan benar bila input yang diberikan akan menghasilkan output yang sesuai dengan spesifikasi program tanpa memperhatikan struktur logika dalam program.

a. Rencana Pengujian

Rencana pengujian yang akan dilakukan pada aplikasi ini dapat dilihat pada Tabel IV.2.

Tabel IV.2 Rencana Pengujian

Kelas Uji	Butir Uji	Jenis Pengujian
Utama	Mengubah konfigurasi Rubik secara manual	Black box
	Mengubah konfigurasi dengan fitur acak	Black box
	Menjalankan algoritma pencarian	Black box
	Menampilkan hasil pencarian	Black box
Petunjuk	Menampilkan petunjuk penggunaan aplikasi	Black box
Tentang Pembuat	Menampilkan informasi pembuat aplikasi	Black box

b. Kasus dan Hasil Pengujian Black Box

Kasus dan hasil pengujian berisi pemaparan dari rencana pengujian yang telah disusun pada scenario pengujian. Pengujian ini dilakukan secara *black box* dengan hanya memperhatikan masukan ke dalam sistem dan keluaran dari masukan tersebut.

1. Pengujian Utama

Pengujian ini dilakukan untuk menguji persyaratan fungsional *Form* Utama yang terdiri dari mengubah konfigurasi Rubik's Cube secara manual, menggunakan fitur acak untuk mengacak konfigurasi, menjalankan algoritma pencarian dan menampilkan hasil pencarian. Kasus dan hasil uji dengan data normal dapat dilihat pada Tabel IV.3.

Tabel IV.3 Kasus dan Hasil Uji(data normal)

Kasus dan hasil uji (data normal)			
Masukan	**Realisasi yang diharapkan**	**Pengamatan**	**Kesimpulan**
Menekan dan menggeser permukaan rubik	Mengubah konfigurasi Rubik berubah sesuai gerakan mouse	Konfigurasi Rubik berubah sesuai gerakan mouse	Diterima
Memasukkan Jumlah Langkah Acak	Dapat terisi dengan masukkan data Jumlah Langkah Acak	Dapat mengisi Jumlah Langkah Acak yang diharapkan	Diterima
Menekan tombol Acak Rubik	Mengacak konfigurasi Rubik dan menampilkan hasilnya sesuai jumlah langkah acak	Tombol dapat berfungsi sesuai yang diharapkan	Diterima
Menekan tombol Reset	Mengembalikan konfigurasi Rubik ke kondisi terselesaikan	Tombol dapat berfungsi sesuai yang diharapkan	Diterima
Memasukkan Jumlah Lebah	Dapat terisi dengan masukkan data Jumlah Lebah	Dapat mengisi Jumlah Lebah yang diharapkan	Diterima
Memasukkan Jumlah Iterasi	Dapat terisi dengan masukkan data Jumlah Iterasi	Dapat mengisi Jumlah Iterasi yang diharapkan	Diterima
Menekan tombol Cari Solusi	Menjalankan algoritma untuk mencari solusi dan menampilkan progressnya ke logbox	Tombol dapat berfungsi sesuai yang diharapkan	Diterima
Menekan tombol Stop Pencarian	Menghentikan proses pencarian	Tombol dapat berfungsi sesuai yang diharapkan	Diterima
Menekan tombol Berikutnya	Mengubah konfigurasi rubik ke langkah solusi berikutnya	Tombol dapat berfungsi sesuai yang diharapkan	Diterima
Menekan tombol Sebelumnya	Mengubah konfigurasi rubik ke langkah solusi sebelumnya	Tombol dapat berfungsi sesuai yang diharakan	Diterima

Kasus dan hasil uji dengan data normal dapat dilihat pada Tabel IV.4.

Tabel IV.4 Kasus dan Hasil Uji(data tidak normal)

Kasus dan hasil uji (data tidak normal)			
Masukan	**Realisasi yang diharapkan**	**Pengamatan**	**Kesimpulan**
Menekan dan menggeser diluar permukaan rubik	Mengabaikan gerakan mouse	Konfigurasi Rubik tidak berubah	Diterima
Menekan tombol Acak Rubik dengan data Jumlah Langkah Acak bukan angka	Menampilkan pesan Jumlah Langkah Acak harus berupa angka	Hasil sesuai dengan yang diharapkan	Diterima
Menekan tombol Acak Rubik dengan data Jumlah Langkah Acak kurang dari 1 atau lebih dari 500	Menampilkan pesan Jumlah Langkah Acak tidak boleh kurang dari 1 atau lebih dari 500	Hasil sesuai dengan yang diharapkan	Diterima
Menekan tombol Cari Solusi dengan data Jumlah Lebah bukan angka	Menampilkan pesan Jumlah Lebah harus berupa angka	Hasil sesuai dengan yang diharapkan	Diterima
Menekan tombol Cari Solusi dengan data Jumlah Lebah kurang dari 12 atau lebih dari 5000	Menampilkan pesan Jumlah Langkah Acak tidak boleh kurang dari 12 atau lebih dari 5000	Hasil sesuai dengan yang diharapkan	Diterima
Menekan tombol Cari Solusi dengan data Jumlah Iterasi bukan berupa angka	Menampilkan pesan Jumlah Iterasi harus berupa angka	Hasil sesuai dengan yang diharapkan	Diterima
Menekan tombol Cari Solusi dengan data Jumlah Iterasi kurang dari 1	Menampilkan pesan Jumlah Iterasi tidak boleh kurang dari 1	Hasil sesuai dengan yang diharapkan	Diterima

2. Pengujian Petunjuk

Pengujian ini dilakukan untuk menguji persyaratan fungsional pemanggilan jendela Petunjuk. Kasus dan hasil ujinya dapat dilihat pada Tabel IV.5.

Tabel IV.5 Kasus dan Hasil Pengujian Petunjuk

Kasus dan hasil uji			
Masukan	**Realisasi yang diharapkan**	**Pengamatan**	**Kesimpulan**
Memilih Petunjuk Penggunaan dari menu Bantuan	Menampilkan Jendela Petunjuk penggunaan aplikasi	Jendela Petunjuk Penggunaan tampil di layar	Diterima

3. Pengujian Tentang Pembuat

Pengujian ini dilakukan untuk menguji persyaratan fungsional pemanggilan jendela Tentang Pembuat. Kasus dan hasi ujinya dapat dilihat pada Tabel IV.6.

Tabel IV.6 Kasus dan Hasil Uji Pengujian Tentang Pembuat

Kasus dan hasil uji			
Masukan	**Realisasi yang diharapkan**	**Pengamatan**	**Kesimpulan**
Memilih Tentang Pembuat dari menu Bantuan	Menampilkan Jendela Tentang pembuat aplikasi	Jendela Tentang Pembuat tampil di layar	Diterima

c. Pengujian Parameter Aplikasi

Parameter aplikasi yang akan diuji yaitu waktu pencarian dan perbandingan panjang langkah acak dengan solusi yang dihasilkan.

Pengujian dilakukan menggunakan beberapa konfigurasi yang dibangkitkan secara acak yang dapat dilihat pada Tabel IV.7.

Tabel IV.7 Konfigurasi Rubik's Cube yang diuji

No	Jumlah Langkah Acak	Langkah Acak Pembangkit
1	5	F F F L R
2	5	FB'DUL
3	6	U' F L' L' D L
4	7	B' U R' F' B R B
5	8	F F U L R' B' D R
6	10	F' B B' F' F' D D L U' D

Pengujian dilakukan dengan jumlah iterasi maksimal 1000 dan jumlah lebah 12, 100, 500 dan 1000. Pada Tabel IV.8 dapat dilihat bahwa untuk kasus nomor 1 dan 6, koloni lebah berhasil menemukan langkah solusi yang lebih pendek dari jumlah langkah operator pembangkitnya.

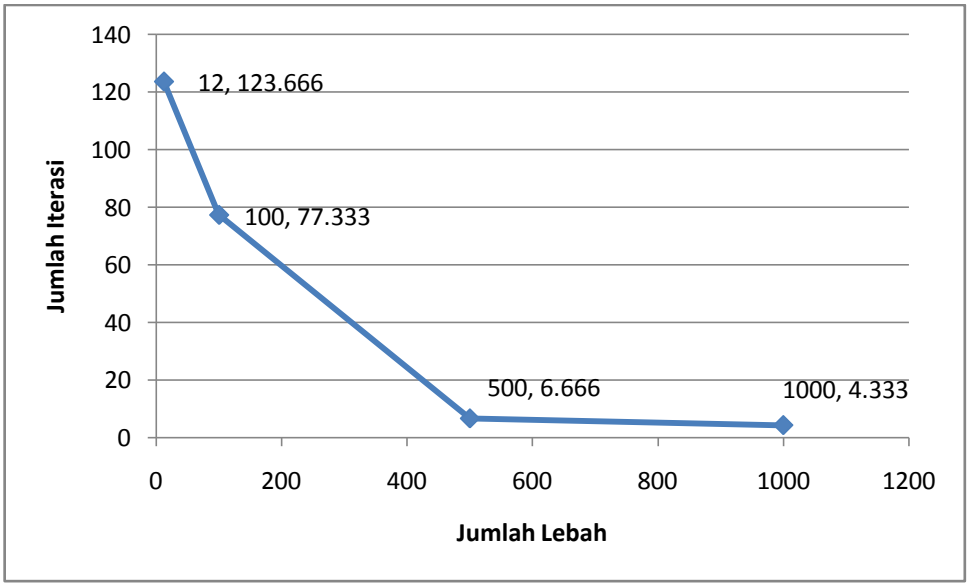
Tabel IV.8 Hasil Pengujian Parameter

No	jumlah lebah	langkah solusi	panjang solusi	jumlah iterasi	jumlah simpul	waktu pencarian	rata-rata waktu per iterasi	rata-rata waktu per kedalaman
1	12	L'R'F	3	2	155	0.075	0.037	0.004
	100	L'R'F	3	1	727	0.019	0.019	0.002
	500	L'R'F	3	1	1288	0.033	0.033	0.004
	1000	R'L'F	3	1	1409	0.04	0.04	0.005
2	12	L'U'D'F'B	5	16	408	0.808	0.050	0.010
	100	L'D'U'BF'	5	2	2630	0.515	0.257	0.051
	500	L'U'D'BF'	5	1	8922	0.088	0.088	0.017
	1000	L'D'U'F'B	5	1	16039	0.332	0.332	0.066
3	12	L'D'L'L'F'U	6	353	529	14.67	0.041	0.006
	100	L'D'L'L'F'U	6	229	3664	72.681	0.317	0.052
	500	L'D'LLF'U	6	18	13102	35.053	1.947	0.324
	1000	L'D'L'L'F'U	6	11	25103	46.348	4.213	0.702
4	12	-		1000	4148			
	100	BR'FB'RU'B	7	562	4423	157.214	0.279	0.039
	500	BR'FB'RU'B	7	82	18173	183.955	2.243	0.320
	1000	BR'FB'RU'B	7	22	31912	73.196	3.327	0.475
5	12	-		1000	4093	42.479	0.042	0.005
	100	R'D'BL'RU'FF	8	53	5105	15.615	0.294	0.036
	500	-		1000	16971	1703.21	1.703	0.212
	1000	R'D'BL'RU'F'F'	8	600	42714	2368.56	3.947	0.493
6	12	UD'L'DDF'	6	965	496	37.628	0.038	0.006
	100	UD'L'D'D'F'	6	5	3290	1.939	0.387	0.064
	500	UD'L'D'D'F'	6	8	12772	17.004	2.125	0.354
	1000	D'UL'D'D'F'	6	1	23783	0.29	0.29	0.048

Dari hasil uji coba di atas memperlihatkan nilai yang berbeda – beda setiap kali algoritma dijalankan. Permasalahan ini disebabkan hampir seluruh proses dalam algoritma BCO mengandalkan teknik acak dengan metode *Roulette Wheel*. Dalam beberapa percobaan, algoritma tidak berhasil menemukan solusi atau terjebak dalam *local optimum*, terutama bila jumlah langkah yang membangkitkan konfigurasi lebih dari 9 dimana terdapat

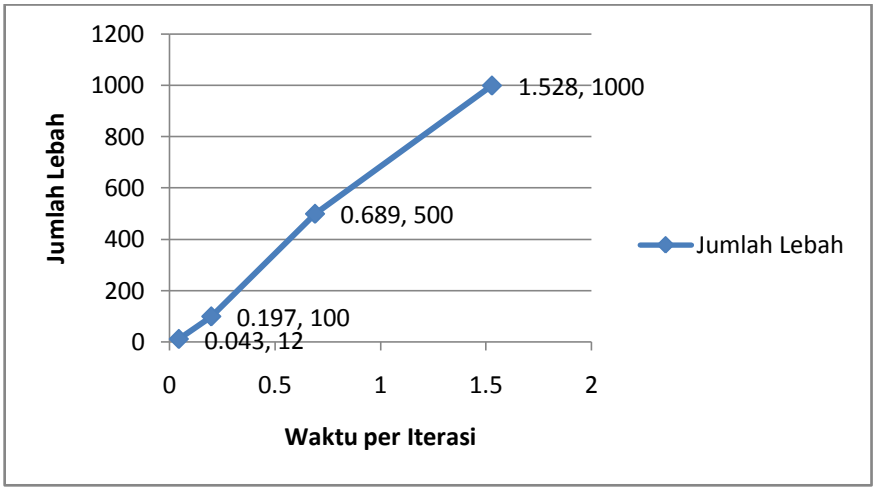
17,596.479.795 konfigurasi yang mungkin dibangkitkan pada kedalaman 9. Jika terjebak *local optimum*, proses akan berjalan terus menerus dan kecil kemungkinan untuk menemukan solusi.

Banyaknya jumlah lebah berbanding terbalik dengan jumlah iterasi yang diperlukan untuk menemukan solusi. Perbandingan antara jumlah iterasi dan jumlah lebah diperlihatkan pada Gambar IV.8.



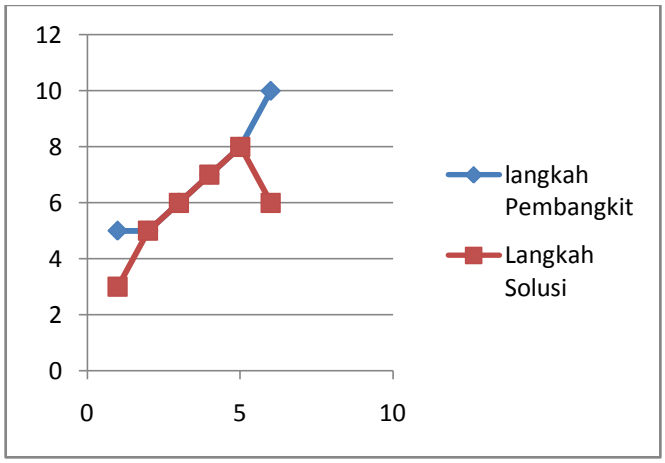
Gambar IV.8 Perbandingan Jumlah Lebah dengan Jumlah Iterasi

Jumlah langkah acak yang membangkitkan konfigurasi Rubik's Cube sangat berpengaruh pada waktu pencarian. Semakin banyak jumlah langkah acak maka waktu pencarian akan semakin lama. Selain itu semakin banyak jumlah lebah, maka waktu pencarian semakin besar. Hal ini dikarenakan jumlah simpul yang harus diproses berbanding lurus dengan jumlah lebah. Gambar IV.9 memperlihatkan perbandingan jumlah lebah dengan waktu pencarian.



Gambar IV.9 Perbandingan Jumlah Lebah dengan Rata-rata Waktu per Iterasi

Dari hasil pengujian dapat disimpulkan pula bahwa Algoritma BCO berhasil menemukan solusi yang lebih pendek dibandingkan dengan langkah yang dibutuhkan untuk membentuk konfigurasi yang dicari solusinya. Hal ini dapat dilihat pada Gambar IV.10

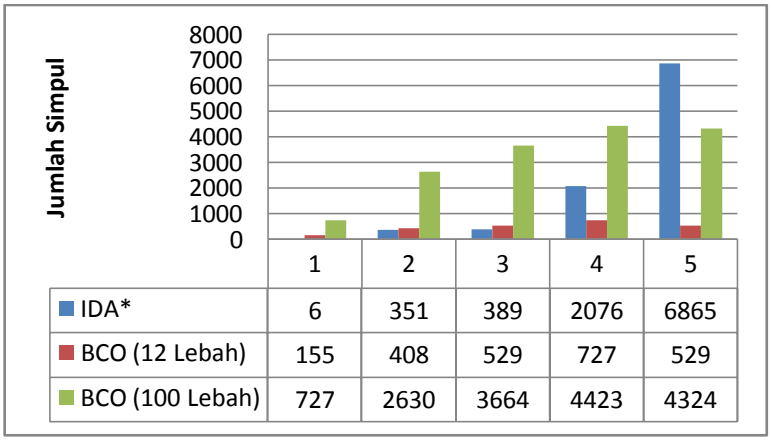


Gambar IV.10 Perbandingan Jumlah Langkah Pembangkit dan Langkah Solusi

Pada pengujian ini juga dilakukan perbandingan metode algoritma BCO dengan metode algoritma IDA*. Parameter yang digunakan sebagai acuan perbandingan adalah jumlah simpul yang ditelusuri dan waktu pencarian.

Jumlah maksimal simpul yang ditelusuri dalam setiap iterasi bersifat linear sesuai dengan jumlah lebah, hal ini lebih efisien bila dibandingkan dengan algoritma IDA* dimana jumlah simpul yang ditelusuri terus meningkat pada setiap kedalaman pencarian.

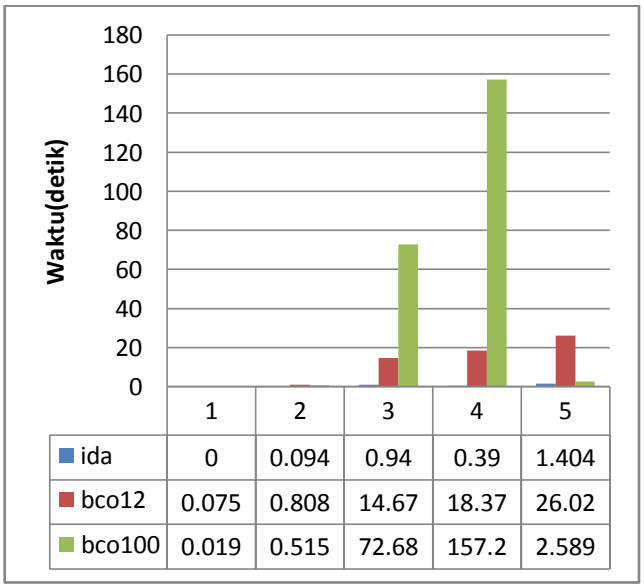
Hasil perbandingan jumlah simpul yang ditelusuri antara algoritma IDA* dan BCO dapat dilihat pada Gambar IV.11.



	1	2	3	4	5
IDA*	6	351	389	2076	6865
BCO (12 Lebah)	155	408	529	727	529
BCO (100 Lebah)	727	2630	3664	4423	4324

Gambar IV.11 Perbandingan Jumlah Simpul yang ditelusuri

Perbandingan waktu pencarian antara algoritma BCO dan IDA* diperlihatkan pada Gambar IV.12.



	1	2	3	4	5
ida	0	0.094	0.94	0.39	1.404
bco12	0.075	0.808	14.67	18.37	26.02
bco100	0.019	0.515	72.68	157.2	2.589

Gambar IV.12 Perbandingan Waktu Pencarian

Dari grafik perbandingan pada Gambar IV.12 dapat dilihat bahwa waktu pencarian algoritma *BCO* cenderung berbeda-beda tetapi cenderung lebih cepat dengan jumlah lebah yang lebih banyak, bila dibandingkan dengan algoritma IDA* dan berbanding terbalik dengan kedalaman solusi yang ditemukan.

IV.3 Kesimpulan pengujian

Dari hasil pengujian dapat disimpulkan pula bahwa Algoritma BCO berhasil menemukan langkah solusi terpendek.

Jumlah langkah acak yang membangkitkan konfigurasi Rubik's Cube sangat berpengaruh pada waktu pencarian. Semakin banyak jumlah langkah acak maka waktu pencarian akan semakin lama. Waktu pencarian cenderung berbeda – beda setiap kali algoritma dijalankan, yang disebabkan hampir seluruh proses dalam algoritma BCO mengandalkan teknik acak dengan metode *Roulette Wheel*. Dalam beberapa percobaan, algoritma tidak berhasil menemukan solusi atau terjebak dalam *local optimum*, sehingga proses akan berjalan terus menerus dan kecil kemungkinan untuk menemukan solusi. Sehingga pada kasus terburuk, performansi Algoritma BCO lebih lama bila dibandingkan dengan algoritma lain, karena terdapat kemungkinan terjebak dalam *local optimum*, yaitu solusi-solusi parsial yang dihasilkan dalam setiap iterasi algoritma BCO.