Tugas Besar 1 IF3170 Inteligensi Artifisial
Pencarian Solusi Diagonal Magic Cube dengan Local Search

Disusun Oleh:
- Adril Putra Merin (13522068)
- Matthew Vladimir Hutabarat (13522093)

PROGRAM STUDI TEKNIK INFORMATIKA

SEKOLAH TEKNIK ELEKTRO DAN INFORMATIKA

INSTITUT TEKNOLOGI BANDUNG

2023

DAFTAR ISI

BAB I
DESKRIPSI PERSOALAN

Diagonal magic cube adalah sebuah kubus dengan ukuran $n \times n \times n$ yang terdiri atas angka-angka dari 1 sampai n^3 tanpa pengulangan, dengan n adalah panjang sisi pada kubus tersebut. *Magic cube* tersebut memiliki n^2 buah baris, n^2 buah kolom, n^2 buah pilar, 6n buah diagonal bidang, dan 4 buah diagonal ruang yang memiliki jumlah elemen sama dengan *magic number*.

Jika sebuah kubus memiliki *magic number*, angka tersebut akan bersifat konstan yang diberikan oleh persamaan berikut:

$$M(n) = \frac{1}{2}n(n^3 + 1)$$

(Eq. 1)

Bukti dari persamaan (1) adalah sebagai berikut. Misalkan sebuah kubus memiliki *magic number* dan sudah tersusun sedemikian rupa sehingga memenuhi syarat untuk disebut *magic cube*. Jumlah setiap elemen dari setiap elemen pada kubus tersebut adalah jumlah angka dari 1 sampai n^3 dimana elemen-elemen tersebut dapat dibagi ke dalam n^2 baris yang memiliki jumlah elemen sama dengan *magic number*. Dengan kata lain,

$$n^2 \times M(n) = \frac{1}{2}n^3(n^3 + 1)$$

$$M(n) = \frac{1}{2}n(n^3 + 1)$$

Pada tugas ini kita akan mencoba menyelesaikan *diagonal magic cube* dengan ukuran $5 \times 5 \times 5$. Initial state dari suatu kubus adalah susunan angka 1 hingga 5^3 secara acak. Kemudian, tiap iterasi pada algoritma *local search*, langkah yang boleh dilakukan adalah menukar posisi dari 2 angka pada kubus tersebut (2 angka yang ditukar tidak harus bersebelahan).

Pada tahun 2003, C. Boyer and W. Trump menemukan *magic cube* dengan ukuran $5 \times 5 \times 5$ sehingga dijamin terdapat solusi yang dapat ditemukan oleh pencarian lokal. Berdasarkan persamaan (1), *magic number* dari kubus dengan ukuran sisi 5 adalah **315**.

Magic cube dengan ukuran sisi 5 memiliki 25 buah baris, 25 buah kolom, 25 buah pilar, 30 buah diagonal bidang, dan 4 buah diagonald ruang. Jadi, secara total, terdapat $25 + 25 + 25 + 30 + 4 = 109$ buah segmen yang harus memiliki jumlah elemen sama dengan *magic cube*.

Pada persoalan ini, setiap state diwakili oleh suatu larik yang berisi permutasi angka 1 hingga 125 yang ditempatkan ke dalam suatu kubus yang berukuran $5 \times 5 \times 5$. Jadi, objektif dari persoalan ini adalah mencari permutasi yang memenuhi setiap properti dari *diagonal magic cube*. Neighbor untuk setiap state didapatkan dengan menukar dua buah elemen pada larik state saat ini. Dengan demikian, setiap state memiliki $125C2 = \frac{1}{2}(125 \times 124)$ = 7750 buah *neighbors*.

BAB II
FUNGSI OBJEKTIF

Fungsi objektif adalah fungsi yang ingin meminimumkan atau memaksimumkan nilainya berdasarkan kendala yang ada. Pada konteks pencarian lokal, fungsi objektif akan mengembalikan nilai dari suatu *state* berdasarkan batasan yang berlaku. Tujuan dari pencarian lokal sendiri adalah untuk memaksimumkan atau meminimumkan hasil fungsi objektif ini.

Pada persoalan ini, batasan yang didefinisikan adalah semua 109 potongan, meliputi baris, kolom, diagonal baris, dan diagonal ruang, harus memiliki jumlah elemen yang sama dengan 315. Kami mengajukan beberapa kandidat fungsi objektif dan akan memilih fungsi objektif terbaik di akhir bab ini.

A. Number of Slices

Kandidat fungsi objektif yang pertama adalah banyak potongan (dari 109 potongan) yang memenuhi batasan, yaitu memiliki jumlah elemen yang sama dengan 315. Pada fungsi objektif ini, tujuan kita adalah memaksimalkan banyak jumlah potongan yang memenuhi batasan tersebut.

Definisikan $h(s)$ sebagai fungsi objektif yang mengembalikan banyak potongan yang memiliki jumlah elemen yang sama dengan 315 pada suatu state. Secara matematis, fungsi objektif ini dapat didefinisikan sebagai berikut:

$$h(s) = \sum_{i=1}^{109} slice(i)$$

dimana $slice(i)$ akan bernilai 1 jika jumlah elemen pada potongan ke-i sama dengan 315, dan bernilai 0 jika sebaliknya. Dengan demikian nilai maksimum dari h adalah $h(s) = 109$. Adapun pseudocode dari fungsi objektif ini adalah sebagai berikut.

```
function OBJECTIVE(state) → integer
  cnt ← 0
  for all slice in state.CUBE:
    if sum_element(slice) = 315 then
      cnt ← cnt + 1
  return cnt
```

Perhatikan bahwa terdapat $125!$ (lebih dari 10^{209}) konfigurasi state yang mungkin, sedangkan banyak kemungkinan nilai h hanyalah 109 buah sehingga suatu state sangat mungkin memiliki nilai h yang sama dengan banyak state lainnya. Akibatnya, pencarian lokal akan memiliki kemungkinan yang sangat besar untuk terjebak pada *local maximum* atau *flat maximum local* (*plateau*). Disamping itu, untuk beberapa algoritma pencarian, suatu *state* akan kesulitan untuk memutuskan *neighbor* yang terbaik untuk dipilih karena banyak *neighbor* dengan nilai h yang sama berjumlah sangat banyak. Hal ini tentu sangat buruk untuk proses pencarian lokal sehingga kami memutuskan tidak menggunakan fungsi objektif ini.

B. Sum of Absolute Differences

Pada kandidat ini, fungsi $g(s)$ didefinisikan sebagai fungsi objektif yang mengembalikan jumlah dari *absolute difference* antara jumlah elemen pada suatu potongan dengan 315 untuk setiap 109 potongan. Tujuan dari pencarian lokal adalah meminimumkan nilai dari fungsi objektif $g(s)$. Secara matematis, fungsi objektif ini dapat didefinisikan sebagai berikut:

$$g(s) = \sum_{i=1}^{109} |315 - sum\_element(i)|$$

dimana $sum\_element(i)$ adalah jumlah semua elemen pada potongan ke-i. Jadi, nilai minimum dari $g(s)$ adalah 0 yang merupakan tujuan utama dari pencarian yang akan dilakukan. Dengan menggunakan fungsi objektif g, kita mencoba mencari susunan kubus yang memiliki *sum of absolute differences* paling kecil. Secara kasar, berikut adalah implementasi fungsi objektif ini dalam pseudocode.

```
function OBJECTIVE(state) → integer
  sum ← 0
  for all slice in state.CUBE:
    sum ← sum + abs(sum_element(i) - 315)
  return sum
```

Perhatikan bahwa, nilai yang dihasilkan oleh fungsi objektif g lebih bervariasi dibandingkan dengan nilai yang dihasilkan oleh fungsi objektif h pada bagian sebelumnya karena kita tidak hanya menghitung banyak potongan kubus yang sudah memenuhi syarat, tetapi juga menghitung jumlah dari perbedaan antara tiap jumlah elemen potongan dengan *magic number*, yaitu 315. Dua buah state yang memiliki $h(s)$ yang sama mungkin saja memiliki nilai $g(s)$ yang berbeda karena alasan diatas. Dengan demikian, kita dapat mengurangi kemungkinan terjebak di *maximum local* dan *flat maximum local* yang sebelumnya dihasilkan oleh $h(s)$.

C. Sum of Squared Differences

Walaupun fungsi objektif $g(s)$ lebih baik daripada $h(s)$, kita masih memiliki kemungkinan yang cukup besar untuk terjebak di *maximum local* dan *plateau*. Untuk nilai *sum of absolute of differences* yang sama, kita ingin agar distribusi *absolute of differences* yang lebih merata, memiliki nilai yang lebih rendah. Namun, perbedaan ini tidak ada pada fungsi objektif $g(s)$. Untuk itu, fungsi objektif yang menjadi alternatif paling baik adalah *sum of squared differences*.

Didefinisikan fungsi $f(s)$ sebagai fungsi objektif yang mengembalikan jumlah dari squared differences antara jumlah elemen dengan 315 untuk setiap potongan. Tujuan dari pencarian lokal adalah meminimumkan nilai dari fungsi objektif $g(s)$. Secara matematis, fungsi objektif ini dapat dituliskan sebagai berikut:

$$f(s) = \sum_{i=1}^{109} (315 - sum\_element(i))^2$$

Adapun implementasi fungsi objektif ini dalam bentuk pseudocode adalah sebagai berikut:

```
function OBJECTIVE(state) → integer
  sum ← 0
  for all slice in state.CUBE:
    sum ← sum + (sum_element(i) - 315) * (sum_element(i) - 315)
  return sum
```

Perbedaan utama antara fungsi objektif ini dengan fungsi objektif sebelumnya (*sum of absolute differences*) terdapat pada kuadrat antara perbedaan jumlah elemen pada potongan dengan 315. Hal ini dilakukan agar perbedaan antara jumlah elemen potongan dengan 315, untuk setiap potongan, yang memiliki distribusi lebih merata, memiliki nilai yang lebih rendah karena tujuan kita adalah minimisasi. Disamping itu, grafik fungsi $f(s)$ bersifat lebih "kontinu" dibandingkan $g(s)$ dan $h(s)$ yang mana lebih baik untuk melakukan pencarian lokal.

Karena alasan yang sudah dijelaskan sebelumnya, kami memutuskan untuk menggunakan *sum of squared differences* sebagai fungsi objektif yang digunakan pada tugas ini.

BAB III
ALGORITMA PENCARIAN LOKAL

Algoritma pencarian lokal adalah algoritma pencarian yang bekerja dengan mencari dari sebuah *start state* ke *neighbouring states* tanpa menyimpan himpunan dari setiap state yang sudah dikunjungi. Hal ini berarti, algoritma ini tidak sistematis karena mungkin saja pencarian tidak akan pernah mengunjungi bagian dari ruang pencarian dimana solusi utama berada. Namun, algoritma ini memiliki dua keuntungan utama dibandingkan *classical search*. Pertama, algoritma ini menggunakan sangat sedikit memori. Kedua, algoritma ini mampu mencari solusi pada ruang pencarian yang besar atau tak hingga dimana pencarian algoritma sistematik (*classical search*) tidak dapat.

Algoritma pencarian lokal juga dapat menyelesaikan persoalan optimasi yang mana tujuannya adalah mencari *state* terbaik berdasarkan suatu fungsi objektif. Karena itu, kita akan menyelesaikan persoalan *diagonal magic cube* dengan menggunakan pencarian lokal. Pertimbangan lain adalah algoritma *classical search* hampir tidak mungkin menemukan solusi yang mungkin karena terdapat 125! states yang perlu dikunjungi.

Selanjutnya, kita akan merancang kelas-kelas dan fungsi-fungsi yang akan digunakan untuk menyelesaikan persoalan ini dalam bahasa Go. Implementasi kelas sebenarnya tidak ada dalam bahasa Go, tetapi untuk mempermudah, kita tetap akan menggunakan konsep kelas untuk menggabungkan fungsi-fungsi yang berhubungan pada suatu entitas. Berikut adalah berbagai kelas dan fungsi yang digunakan dalam implementasi algoritma pencarian lokal.

1. Kelas State

   Kelas state akan menyimpan larik sepanjang 125 yang merepresentasikan state dari sebuah kubus. Kelas ini juga akan menyimpan *objective value* saat ini dan jumlah dari setiap elemen dari 109 *slices* yang ada untuk mempercepat proses kalkulasi . Selain itu, kelas state juga akan menyimpan beberapa fungsi utilitas yang akan digunakan untuk keperluan pencarian lokal nantinya. Kelas state ini akan digunakan untuk algoritma pencarian lokal seperti *steepest ascent hill climbing*, *hill climbing with sideways move*, *hill-climbing with random restart*, *stochastic hill climbing*, *simulated annealing,* dan *genetic algorithm*.

   A. Definisi Tipe

   Seperti yang sudah dibahas sebelumnya, sebuah **State** terdiri atas *array of integer* sepanjang 125 yang menyimpan susunan angka pada kubus, jumlah dari setiap elemen dari 109 *slices*, dan *objective value* saat ini.

```
type State struct {
      Cubes   []int
      CurrSum []int
      Value   int
}
```

   B. Fungsi InitVal

   Fungsi ini adalah fungsi yang menghitung jumlah dari setiap elemen dari 109 *slices* yang ada dan nilai dari *objective value* berdasarkan susunan angka pada kubus.

```
func (state *State) InitVal() {
	state.CurrSum = make([]int, 109)
	// row
	idx := 0
	for i := 0; i < 5; i++ {
		for j := 0; j < 5; j++ {
			sum := 0
			for k := 0; k < 5; k++ {
				absIdx := i*5*5 + j*5 + k
				sum += state.Cubes[absIdx]
			}
			state.CurrSum[idx] = sum
			idx++
		}
	}

	// column
	for i := 0; i < 5; i++ {
		for k := 0; k < 5; k++ {
			sum := 0
			for j := 0; j < 5; j++ {
				absIdx := i*5*5 + j*5 + k
				sum += state.Cubes[absIdx]
			}
			state.CurrSum[idx] = sum
			idx++
		}
	}

	// tiang
	for j := 0; j < 5; j++ {
		for k := 0; k < 5; k++ {
			sum := 0
			for i := 0; i < 5; i++ {
				absIdx := i*5*5 + j*5 + k
				sum += state.Cubes[absIdx]
			}
			state.CurrSum[idx] = sum
			idx++
		}
	}

	// calculate the first diagonal plane
	for i := 0; i < 5; i++ {
		sum1 := 0
		sum2 := 0
		for j := 0; j < 5; j++ {
			absIdx1 := i*5*5 + j*5 + j
			absIdx2 := i*5*5 + j*5 + (4 - j)
			sum1 += state.Cubes[absIdx1]
			sum2 += state.Cubes[absIdx2]
		}
		state.CurrSum[idx] = sum1
		idx++
		state.CurrSum[idx] = sum2
```

```
            idx++
      }

      // calculate the second diagonal plane
      for j := 0; j < 5; j++ {
            sum1 := 0
            sum2 := 0
            for i := 0; i < 5; i++ {
                  absIdx1 := i*5*5 + j*5 + i
                  absIdx2 := i*5*5 + j*5 + (4 - i)
                  sum1 += state.Cubes[absIdx1]
                  sum2 += state.Cubes[absIdx2]
            }
            state.CurrSum[idx] = sum1
            idx++
            state.CurrSum[idx] = sum2
            idx++
      }

      // calculate the third diagonal plane
      for k := 0; k < 5; k++ {
            sum1 := 0
            sum2 := 0
            for j := 0; j < 5; j++ {
                  absIdx1 := j*5*5 + j*5 + k
                  absIdx2 := (4-j)*5*5 + j*5 + k
                  sum1 += state.Cubes[absIdx1]
                  sum2 += state.Cubes[absIdx2]
            }
            state.CurrSum[idx] = sum1
            idx++
            state.CurrSum[idx] = sum2
            idx++
      }

      // space diagonal
      sum1 := 0
      sum2 := 0
      sum3 := 0
      sum4 := 0
      for i := 0; i < 5; i++ {
            absIdx1 := i*5*5 + i*5 + i
            absIdx2 := i*5*5 + (4-i)*5 + i
            absIdx3 := (4-i)*5*5 + i*5 + i
            absIdx4 := (4-i)*5*5 + (4-i)*5 + i
            sum1 += state.Cubes[absIdx1]
            sum2 += state.Cubes[absIdx2]
            sum3 += state.Cubes[absIdx3]
            sum4 += state.Cubes[absIdx4]
      }
      state.CurrSum[idx] = sum1
      idx++
      state.CurrSum[idx] = sum2
      idx++
      state.CurrSum[idx] = sum3
```

```
	idx++
	state.CurrSum[idx] = sum4
	idx++

	res := 0
	for i := 0; i < 109; i++ {
		res += ((315 - state.CurrSum[i]) * (315 - state.CurrSum[i]))
	}
	state.Value = res
}
```

C. Fungsi InitEmptyCubes

Fungsi yang menginisialisasi susunan angka pada kubus menjadi kosong.

```
func (state *State) InitEmptyCubes() {
	state.Cubes = make([]int, 125)
}
```

D. Fungsi Init

Fungsi yang menginisialisasi sebuah **State** dengan susunan angka yang acak. Fungsi ini juga memanggil InitVal untuk melakukan kalkulasi terhadap Value dan CurrSum.

```
func (state *State) Init() {
	state.Cubes = []int{}
	for i := 1; i <= 125; i++ {
		state.Cubes = append(state.Cubes, i)
	}
	rand.Shuffle(len(state.Cubes), func(i, j int) { state.Cubes[i],
state.Cubes[j] = state.Cubes[j], state.Cubes[i] })
	state.InitVal()
}
```

E. Fungsi getAffectedArea

Fungsi yang mengembalikan indeks dari baris, kolom, diagonal bidang, dan diagonal ruang yang terpengaruh oleh suatu elemen pada larik kubus.

```
func getAffectedArea(idx int) []int {
	res := []int{}
	i := idx / 25
	j := (idx % 25) / 5
	k := idx % 5

	// check row
	idxRow := 5*i + j
	res = append(res, idxRow)

	// check col
	idxCol := 25 + 5*i + k
	res = append(res, idxCol)

	// check tiang
```

```
	idxTiang := 50 + 5*j + k
	res = append(res, idxTiang)

	// check diagonal-1
	if j == k {
		res = append(res, 75+i*2)
	} else {
		res = append(res, -1)
	}

	if k == 4-j {
		res = append(res, 75+i*2+1)
	} else {
		res = append(res, -1)
	}

	// check diagonal-2
	if i == k {
		res = append(res, 85+j*2)
	} else {
		res = append(res, -1)
	}

	if k == 4-i {
		res = append(res, 85+j*2+1)
	} else {
		res = append(res, -1)
	}

	// check diagonal-3
	if i == j {
		res = append(res, 95+k*2)
	} else {
		res = append(res, -1)
	}

	if i == 4-j {
		res = append(res, 95+k*2+1)
	} else {
		res = append(res, -1)
	}

	// check space diagonal

	if i == j && j == k {
		res = append(res, 105)
	} else {
		res = append(res, -1)
	}
	if j == 4-i && i == k {
		res = append(res, 106)
	} else {
		res = append(res, -1)
	}
	if j == k && i == 4-j {
```

```
            res = append(res, 107)
      } else {
            res = append(res, -1)
      }
      if i == j && j == 4-k {
            res = append(res, 108)
      } else {
            res = append(res, -1)
      }
      return res
}
```

F. Fungsi BestNeighbor

Fungsi yang mengembalikan *neighbor* dengan *objective value* paling tinggi. Fungsi ini bekerja dengan menukar dua buah elemen untuk setiap pasangan indeks (i, j) dengan $0 \leq i < j < 125$ dan mencari penukaran yang menghasilkan *value* paling rendah.

```
func (state *State) BestNeighbor() (*State, int, int) {
      minVal := 1000000
      first := -1
      second := -1
      arrayFirst := []int{}
      arraySecond := []int{}
      for i := 0; i < 125; i++ {
            for j := i + 1; j < 125; j++ {
                  currVal := int(state.Value)
                  affectedArea1 := getAffectedArea(i)
                  affectedArea2 := getAffectedArea(j)

                  for k := 0; k < 13; k++ {
                        area1 := affectedArea1[k]
                        area2 := affectedArea2[k]
                        if area1 != area2 {
                              if area1 != -1 {
                                    currVal -= (315 -
state.CurrSum[area1]) * (315 - state.CurrSum[area1])
                                    currVal += (315 -
(state.CurrSum[area1] + state.Cubes[j] - state.Cubes[i])) * (315 -
(state.CurrSum[area1] + state.Cubes[j] - state.Cubes[i]))
                              }
                              if area2 != -1 {
                                    currVal -= (315 -
state.CurrSum[area2]) * (315 - state.CurrSum[area2])
                                    currVal += (315 -
(state.CurrSum[area2] + state.Cubes[i] - state.Cubes[j])) * (315 -
(state.CurrSum[area2] + state.Cubes[i] - state.Cubes[j]))
                              }
                        }
                  }

                  if currVal < minVal {
                        minVal = currVal
                        arrayFirst = []int{}
```

```
                        arraySecond = []int{}
                        arrayFirst = append(arrayFirst, i)
                        arraySecond = append(arraySecond, j)
                  } else if currVal == minVal {
                        arrayFirst = append(arrayFirst, i)
                        arraySecond = append(arraySecond, j)
                  }
            }
      }

      rndFirst := rand.Intn(len(arrayFirst))
      first = arrayFirst[rndFirst]
      second = arraySecond[rndFirst]

      newState := State{Value: minVal}
      newState.Cubes = make([]int, len(state.Cubes))
      copy(newState.Cubes, state.Cubes)
      newState.CurrSum = make([]int, len(state.CurrSum))
      copy(newState.CurrSum, state.CurrSum)

      affectedArea1 := getAffectedArea(first)
      affectedArea2 := getAffectedArea(second)

      for k := 0; k < 13; k++ {
            area1 := affectedArea1[k]
            area2 := affectedArea2[k]
            if area1 != area2 {
                  if area1 != -1 {
                        newState.CurrSum[area1] = newState.CurrSum[area1]
+ newState.Cubes[second] - newState.Cubes[first]
                  }
                  if area2 != -1 {
                        newState.CurrSum[area2] = newState.CurrSum[area2]
+ newState.Cubes[first] - newState.Cubes[second]
                  }
            }
      }

      temp := newState.Cubes[first]
      newState.Cubes[first] = newState.Cubes[second]
      newState.Cubes[second] = temp
      return &newState, first, second
}
```

G. Fungs RandomNeighbor

Fungsi yang mengembalikan *neighbor* acak dari *state* saat ini. Algoritma ini bekerja dengan cara memilih pasangan (i, j) secara acak dan menukar elemen pada pasangan indeks tersebut.

```
func (state *State) RandomNeighbor() (*State, int, int) {
      first := rand.Intn(125)
      second := rand.Intn(125)

      for first == second {
            second = rand.Intn(125)
```

```
	}

	newState := State{}
	newState.Cubes = make([]int, len(state.Cubes))
	copy(newState.Cubes, state.Cubes)
	newState.CurrSum = make([]int, len(state.CurrSum))
	copy(newState.CurrSum, state.CurrSum)

	currVal := state.Value
	affectedArea1 := getAffectedArea(first)
	affectedArea2 := getAffectedArea(second)

	for k := 0; k < 13; k++ {
		area1 := affectedArea1[k]
		area2 := affectedArea2[k]
		if area1 != area2 {
			if area1 != -1 {
				currVal -= (315 - state.CurrSum[area1]) * (315 -
state.CurrSum[area1])
				currVal += (315 - (state.CurrSum[area1] +
state.Cubes[second] - state.Cubes[first])) * (315 - (state.CurrSum[area1]
+ state.Cubes[second] - state.Cubes[first]))
				newState.CurrSum[area1] = newState.CurrSum[area1]
+ newState.Cubes[second] - newState.Cubes[first]
			}
			if area2 != -1 {
				currVal -= (315 - state.CurrSum[area2]) * (315 -
state.CurrSum[area2])
				currVal += (315 - (state.CurrSum[area2] +
state.Cubes[first] - state.Cubes[second])) * (315 - (state.CurrSum[area2]
+ state.Cubes[first] - state.Cubes[second]))
				newState.CurrSum[area2] = newState.CurrSum[area2]
+ newState.Cubes[first] - newState.Cubes[second]
			}
		}
	}

	temp := newState.Cubes[first]
	newState.Cubes[first] = newState.Cubes[second]
	newState.Cubes[second] = temp

	newState.Value = currVal
	return &newState, first, second
}
```

A. Steepest Ascent Hill-Climbing

Algoritma Steepest Ascent Hill-Climbing melakukan iterasi terus menerus mencari *neighbor* dengan fungsi objektif dengan nilai yang paling rendah, jika hasil fungsi objektif dari state *neighbor* sama atau lebih dari fungsi objektif state *current* maka akan mengembalikan state *current*. Jika nilainya kurang dari fungsi objektif state *current* maka perbaharui variabel *current* dengan state *neighbor*.

```
func HillClimbing() (*models.State, *models.State, []models.Iteration, int) {
	initial := &models.State{}
	initial.Init()

	current := initial
	iterations := []models.Iteration{}
	iter := 0

	for {
		iter++
		neighbor, first, second := current.BestNeighbor()

		if neighbor.Value >= current.Value {
			return initial, current, iterations, iter
		}

		current = neighbor
		iterations = append(iterations, models.Iteration{
			First:  first,
			Second: second,
			Value:  current.Value,
			Exp:    1,
		})
	}
}
```

Kelebihan	Kekurangan
Karena selalu memilih neighbor dengan nilai objektif yang lebih baik maka akan menyelesaikan problem lebih cepat (walaupun tidak global optimum)	Sering sekali terjebak di local optima karena tidak punya alternatif lain dalam memilih neighbor selain memilih nilai objektif yang lebih tinggi

B. Hill-climbing with Sideways Move

Algoritma Hill-Climbing with Sideways Move melakukan iterasi terus menerus mencari *neighbor* dengan fungsi objektif dengan nilai yang paling rendah, jika hasil fungsi objektif dari state *neighbor* lebih dari fungsi objektif state *current* maka akan mengembalikan state *current*. Jika nilainya kurang dari atau sama dengan fungsi objektif state *current* maka perbaharui variabel *current* dengan state *neighbor*. Algoritma ini memiliki kemungkinan untuk terus menerus sideway tanpa batas sehingga membuat program terjebak dalam *infinity loop*. Untuk menghindari hal ini, diperlukan pembatasan seberapa banyak langkah sideway yang terjadi.

```
func HillClimbingSideways(maxMove int) (*models.State, *models.State,
[]models.Iteration, int) {
	initial := &models.State{}
	initial.Init()

	current := initial
	iterations := []models.Iteration{}
```

```
	iter := 0
	counterMove := 0

	for {
		iter++
		neighbor, first, second := current.BestNeighbor()

		if neighbor.Value > current.Value {
			return initial, current, iterations, iter
		} else if neighbor.Value == current.Value {
			if counterMove < maxMove {
				counterMove++
				current = neighbor
				iterations = append(iterations, models.Iteration{
					First:  first,
					Second: second,
					Value:  current.Value,
					Exp:    1,
				})
			} else {
				return initial, current, iterations, iter
			}
		} else {
			counterMove = 0
			current = neighbor
			iterations = append(iterations, models.Iteration{
				First:  first,
				Second: second,
				Value:  current.Value,
				Exp:    1,
			})
		}
	}
}
```

Kelebihan	Kekurangan
Opsi memilih neighbor dengan nilai objektif yang sama membuat kemungkinan mencapai global optimum lebih tinggi daripada steepest ascent	Tanpa mekanisme untuk membatasi langkah sideways, program akan terus menerus bergerak di local optima tanpa adanya kemajuan
	Masih terdapat kemungkinan untuk stuck di local optima karena tidak ada opsi lain untuk memilih *neighbor* selain nilai objektif lebih tinggi atau sama dari *current*

C. Hill-climbing with Random Restart

Algoritma hill-climbing with random restart bekerja dengan melakukan serangkaian percobaan pencarian hill-climbing dengan initial state yang random hingga solusi tujuan ditemukan. Untuk setiap

hill climbing, cara kerjanya sama dengan *steepest ascent hill climbing*, yaitu untuk setiap state akan dibangkitkan tetangga yang terbaik dan berpindah ke tetangga tersebut jika memiliki nilai yang lebih optimal dibandingkan nilai state saat ini. Berikut adalah implementasi dari algoritma ini.

```
func RandomRestartHillClimbing(maxRestart int) (*models.State, *models.State,
[]models.RestartIteration, int) {
	restartIteration := []models.RestartIteration{}
	counter := 0

	best := &models.State{}
	best.Init()
	best.Value = 10000000

	for counter < maxRestart {
		counter++
		init, final, iteration, iter := HillClimbing()
		restartIteration = append(restartIteration, models.RestartIteration{
			Initial: *init,
			Final:   *final,
			Iter:    iteration,
			NumIter: iter,
		})
		if final.Value < best.Value {
			best = final
		}
		if best.Value == 0 {
			break
		}
	}

	return &restartIteration[0].Initial, best, restartIteration, counter
}
```

Kelebihan	Kekurangan
Dengan berulang kali melakukan pencarian, pada akhirnya kita akan menemukan solusi *global optima* jika *initial state* memiliki nilai yang baik.	Tidak efisien untuk permasalahan yang *simpl*e karena *random restart* mungkin akan tidak diperlukan dan tidak efisien.
Karena berulang kali melakukan pencarian hill climbing, algoritma ini terhindar dari lokal maksimum karena dapat mengeksplor area lain melalui restart yang dilakukan pada initial state yang acak.	Setiap *restart* akan memulai pencarian yang baru dan mungkin saja nilai objektifnya menjadi sangat jauh dari nilai optimal sehingga akan memerlukan waktu yang lama untuk menyelesaikan persoalan ketimbang sideway dan steepest ascent

D. Stochastic Hill-Climbing

Algoritma selanjutnya yang akan dibahas adalah *stochastic hill-climbing*. Algoritma ini bekerja dengan memilih tetangga acak dari *state* saat ini. Jika nilai fungsi objektif dari tetangga tersebut lebih kecil dibandingkan nilai dari *state* saat ini, kita akan berpindah ke state tersebut. Jika tidak, kita akan melanjutkan iterasi. Iterasi akan diterminasi jika sudah mencapai NMAX kali. Algoritma ini akan *converge* lebih lambat dibandingkan algoritma *steepest ascent*, tetapi pada beberapa kasus, algoritma ini dapat menemukan solusi yang baik karena tidak akan terjebak dalam *local optima*. Berikut adalah implementasi dari algoritma ini.

```
func StochasticHillClimbing() (*models.State, *models.State, []models.Iteration,
int) {
	initial := &models.State{}
	initial.Init()

	current := initial
	iterations := []models.Iteration{}
	counter := 0

	for counter < 800000 {
		counter++
		neighbor, first, second := current.RandomNeighbor()

		if neighbor.Value < current.Value {
			current = neighbor
			iterations = append(iterations, models.Iteration{
				First:  first,
				Second: second,
				Value:  current.Value,
				Exp:    1,
			})
		}
		if current.Value == 0 {
			return initial, current, iterations, counter
		}
	}
	return initial, current, iterations, counter
}
```

Kelebihan	Kekurangan
Tidak seperti *hill-climbing* lainnya yang deterministik, *stochastic hill climbing* mengambil tetangga secara acak. Hal ini membantu menghindari jebakan *local optima* dengan memberi kesempatan untuk mengeksplorasi solusi	Meskipun *stochastic hill climbing* dapat menghindari *local optima* dalam beberapa kasus, tidak ada jaminan bahwa algoritma ini akan mencapai solusi optimal secara global.
Karena adanya elemen acak, algoritma ini dapat menjelajahi lebih banyak solusi yang berpotensi	Karena ada elemen acak, algoritma ini mungkin saja mengeksplorasi terlalu banyak solusi yang

lebih baik daripada algoritma *hill climbing* lain yang deterministik.	kurang relevan. Akibatnya, algoritma ini mengalami *converge* yang lebih lambat dibandingkan algoritma *steepest ascent* dan meningkatkan inefisiensi.

E. Simulated Annealing

Algoritma *hill-climbing* yang selalu berpindah ke *state* dengan nilai fungsi objektif lebih tinggi sangat rawan untuk terjebak pada *local optima*. Di sisi lain, algoritma yang bergerak hanya berdasarkan tetangga yang dibangkitkan secara acak tanpa memperhatikan nilai fungsi objektif pada akhirnya akan mendapatkan nilai *global optima*, tetapi akan sangat tidak efisien. Salah satu cara untuk menggabungkan *hill climbing* dengan gerakan acak untuk menghasilkan solusi yang efisien dan *complete* adalah dengan *simulated annealing*.

Simulated annealing terinspirasi dari proses *annealing* di bidang metalurgi, yaitu proses yang digunakan untuk mengeraskan logam dan kaca dengan cara memanaskannya dengan temperatur tinggi dan kemudian mendinginkannya secara perlahan. Ide utama dari *simulated annealing* adalah membiarkan beberapa gerakan yang buruk, tetapi secara perlahan mengurangi frekuensinya. Secara umum, algoritma *simulated annealing* bekerja sebagai berikut untuk persoalan minimasi.

1. Pada awalnya, kita diberikan suatu *state* acak.
2. Pada setiap iterasi, tetangga dari *state* saat ini akan dibangkitkan secara acak.
3. Jika nilai fungsi objektif dari tetangga lebih kecil dari *state* saat ini, *state* akan berpindah ke tetangga.
4. Jika nilai fungsi objektif dari tetangga tidak lebih kecil dari *state* saat ini, algoritma ini memungkinkan *state* untuk berpindah ke tetangga dengan probabilitas tertentu. Probabilitas ini dihitung menggunakan fungsi eksponensial yang bergantung pada perbedaan nilai objektif antara tetangga dan *state*, dan suhu saat ini:

   $$P = exp(\frac{\Delta E}{T})$$

   dimana:
   - $\Delta E = OBJECTIVE(CURR) - OBJECTIVE(NEIGHBOR)$
   - T adalah suhu pada iterasi tersebut yang akan berangsur-angsur berkurang
5. Suhu diturunkan secara bertahap menurut suatu skema pendinginan sehingga awalnya banyak *state* "buruk" yang diterima, tetapi seiring waktu frekuensinya berkurang. Adapun penjadwalan suhu yang digunakan adalah penjadwalan eksponensial.

   $$T = T_0 \times \alpha^t$$

   dengan $T_0 = 9000000$ dan $\alpha = 0.9995$.
6. Algoritma ini berhenti ketika suhu sudah mencapai titik 0 ($T = 0$) atau solusi global sudah ditemukan.

Berdasarkan deskripsi tersebut, kita dapat merancang solusi *simulated annealing* untuk menemukan solusi *diagonal magic cube*. Berikut adalah impelementasi *simulated annealing* untuk menyelesaikan *diagonal magic cube*.

```
func SimulatedAnnealing() (*models.State, *models.State, []models.Iteration,
int, int) {
	initial := &models.State{}
	initial.Init()

	current := initial
	iterations := []models.Iteration{}
	iter := 0
	stuck := 0

	for t := 1; ; t++ {
		iter++
		T := schedule(t)

		if T <= 0.01 || current.Value == 0 {
			return initial, current, iterations, iter, stuck
		}

		neighbor, first, second := current.RandomNeighbor()

		deltaE := current.Value - neighbor.Value

		if deltaE > 0 {
			current = neighbor
			iterations = append(iterations, models.Iteration{
				First:  first,
				Second: second,
				Value:  current.Value,
				Exp:    1,
			})
		} else {
			probability := math.Exp(float64(deltaE) / T)
			if rand.Float64() <= probability {
				current = neighbor
				stuck++
				iterations = append(iterations, models.Iteration{
					First:  first,
					Second: second,
					Value:  current.Value,
					Exp:    probability,
				})
			}
		}
	}
}

func schedule(t int) float64 {
	initialTemperature := 9000000.0
	coolingRate := 0.99995
```

```
        T := initialTemperature * math.Pow(coolingRate, float64(t))

        return T
}
```

F. Genetic Algorithm

Genetic Algorithm (GA) adalah metode untuk memecahkan masalah optimasi, baik *constrained* maupun *unconstrained* berdasarkan proses seleksi dalam dalam biologi evolusi. Secara umum, algoritma ini memodifikasi populasi dari *state* berulang kali. Pada setiap langkah, *genetic algorithm* memilih individu-individu dari populasi saat ini dan menggunakannya sebagai induk untuk memproduksi anak-anak untuk generasi berikutnya. Dari generasi ke generasi, populasi "berevolusi" untuk menemukan solusi yang optimal. Untuk lebih mengerti tentang algoritma ini, berikut adalah beberapa terminologi penting dalam *genetic algorithm*.

1. **Populasi**: GA bekerja dengan sekelompok solusi potensial yang disebut populasi. Setiap populasi disebut individu (state). Pada awalnya, populasi ini diinisialisasi secara acak.
2. **Fitness Function**: Setiap individu (state), dievaluasi menggunakan fungsi yang disebut fitness function, yang mengukur seberapa baik individu tersebut dalam menyelesaikan masalah atau mendekati solusi optimal.
3. **Selection**: Individu yang memiliki nilai fitness yang lebih tinggi memiliki peluang lebih besar untuk dipilih untuk menghasilkan keturunan (offspring). Teknik seleksi yang umum dilakukan antara lain: *roulette wheel selection*, *tournament rank*, dan *rank-based selection*.
4. **Crossover**: Dua individu dipilih dari populasi dan digabungkan untuk menghasilkan keturunan baru. Crossover adalah proses yang mirip dengan reproduksi seksual dalam biologi, di mana sebagian dari materi genetik orang tua dipertukarkan.
5. **Mutation**: Setelah crossover, beberapa individu mengalami mutasi, yaitu perubahan acak pada kromosom. Mutasi membuat keragaman dalam populasi dan mencegah GA terjebak dalam solusi lokal.

Pada persoalan *diagonal magic cube*, kita menggunakan fungsi objektif yang ingin diminimasi, sedangkan *fitness function* dalam konteks GA dirancang untuk memaksimalkan nilainya. Untuk itu, kita perlu merancang *fitness function* untuk mengkonversi persoalan minimasi menjadi persoalan maksimasi. Pendekatan yang dipilih adalah *inverse objective function* sehingga *fitness function* didefinisikan sebagai berikut.

$$fitness(state) = \frac{1}{1+OBJECTIVE(state)}$$

Fungsi tersebut memastikan bahwa semakin kecil nilai OBJECTIVE(state), semakin besar pula nilai *fitness*-nya.

Untuk algoritma *selection*, kita akan memilih *roulette wheel selection* dimana probabilitas tiap individu untuk dipilih didefinisikan sebagai nilai *fitness* suatu individu dibagi dengan nilai *fitness* total untuk setiap individu dalam populasi. Persentase *roulette wheel* akan disusun dimulai dari individu dengan persentase paling tinggi. Dengan demikian, individu dengan nilai *fitness* lebih tinggi memiliki probabilitas yang lebih tinggi pula untuk dipilih.

Proses *crossover* membutuhkan sebuah algoritma yang dapat merekombinasi dua buah individu menjadi individu baru yang tidak kehilangan sifat permutasinya (memiliki angka unik dari 1 - 125). Untuk itu, kami memutuskan untuk menggunakan algoritma *order crossover*. Berikut adalah cara kerja algoritma ini secara singkat.

1. Misalkan terdapat dua buah individu dalam populasi yang ingin direkombinasi, yaitu P_0 dan P_1.

$$P_0 = (A, B, C, D, E, F, G, H, I, J)$$

$$P_1 = (B, D, A, H, J, C, E, G, F, I)$$

2. Pilih beberapa segment acak pada P_0.

$$P_0 = (\underline{A, B}, C, D, E, \underline{F, G, H}, I, J)$$

3. Copy segment yang terpilih dari P_0 ke individu anak (hasil).

$$P_C = (A, B, ?, ?, ?, F, G, H, ?, ?)$$

4. Sisa elemen yang belum ada pada anak, ditransfer dari P_1 sesuai dengan urutan kemunculannya pada individu P_1.

$$P_{\text{missing}} = \{C, D, E, I, J\}$$

$$P_{\text{in order from } P_1} = (D, J, C, E, I)$$

5. Hasil dari rekombinasi tersebut akan menghasilkan individu baru.

$$P_C = (A, B, \underline{D, J, C}, F, G, H, \underline{E, I})$$

Tahap terakhir yang perlu diperhatikan adalah mutasi. Sama halnya seperti *crossover*, mutasi juga memerlukan sebuah algoritma yang dapat memastikan bahwa individu yang dimutasi tetap mempertahankan sifat permutasi yang dimilikinya. Algoritma yang kita pilih adalah *rotation to the right*. Berikut adalah cara kerja algoritma ini secara singkat.

1. Misalkan kita memiliki sebuah individu yang berada dalam bentuk permutasi, misalkan P_0.
2. Pilih sebuah *list* parsial dengan menentukan sebuah index awal *i* dan indeks akhir *j* pada P_0, yang memenuhi $0 < i, j < |P_0 - 1|$. Perhatikan bahwa, *i* dan *j* dapat bersifat sirkular.
3. Tentukan bilangan *k* yang menyatakan banyak rotasi dilakukan pada *list* parsial tersebut.
4. Lakukan rotasi pada *sublist* tersebut sebanyak *k* kali ke kanan.

Selanjutnya, kita akan mendesain implementasi dari GA untuk menyelesaikan persoalan *diagonal magic cube* berdasarkan algoritma-algoritma yang sudah dibahas sebelumnya. Berikut adalah rancangan kelas dan implementasi dari algoritma GA dalam pseudo code

```
func GeneticAlgorithm(maxIteration int, initPopulation int) ([]*models.State,
[]*models.State, []models.GeneticIteration, *models.State, int) {
      initialPops := []*models.State{}
```

```
	for i := 0; i < initPopulation; i++ {
		newState := models.State{}
		newState.Init()
		initialPops = append(initialPops, &newState)
	}

	iterations := []models.GeneticIteration{}

	valSum := 0
	minVal := 10000000
	minIdx := 0
	population := initialPops
	for iter := 1; iter <= maxIteration; iter++ {
		newPopulation := []*models.State{}
		valSum = 0
		minVal = 10000000
		minIdx = 0
		for i := 0; i < len(population); i++ {
			x := randomSelection(population)
			y := randomSelection(population)
			child := reproduce(x, y)
			if rand.Float64() < 0.1 {
				mutation(child)
			}
			valSum += child.Value
			if minVal > child.Value {
				minVal = child.Value
				minIdx = i
			}
			newPopulation = append(newPopulation, child)
		}
		population = newPopulation
		avgVal := float64(valSum) / float64(len(population))
		iterations = append(iterations, models.GeneticIteration{MinValue:
minVal, AvgValue: avgVal})

		if minVal == 0 {
			return initialPops, population, iterations,
population[minIdx], iter
		}
	}
	return initialPops, population, iterations, population[minIdx],
maxIteration
}

func fitness(state *models.State) float64 {
	return 1.0 / (1.0 + float64(state.Value))
}

func randomSelection(population []*models.State) *models.State {
	prefSum := []float64{0}
	sum := 0.0
	for i := 0; i < len(population); i++ {
		fit := fitness(population[i]) * 1000
		sum += fit
```

```
            prev := prefSum[i]
            prefSum = append(prefSum, prev+fit)
      }

      selectedVal := rand.Float64() * sum

      lo := 0
      hi := len(prefSum)
      ans := -1
      for lo <= hi {
            mid := (lo + hi) / 2
            if prefSum[mid] <= selectedVal {
                  ans = mid
                  lo = mid + 1
            } else {
                  hi = mid - 1
            }
      }

      return population[ans]
}

func reproduce(p1 *models.State, p2 *models.State) *models.State {
      child := models.State{}
      vis := make([]bool, 125)
      child.InitEmptyCubes()
      start, end := randomSegment()
      rangeLen := (end - start + 1 + 125) % 125
      for i := 0; i < rangeLen; i++ {
            child.Cubes[(start+i)%125] = p1.Cubes[(start+i)%125]
            vis[child.Cubes[(start+i)%125]-1] = true
      }

      i := 0
      j := 0

      for i < 125 && j < 125 {
            if child.Cubes[i] != 0 {
                  i++
                  continue
            }
            if vis[p2.Cubes[j]-1] {
                  j++
                  continue
            }
            child.Cubes[i] = p2.Cubes[j]
            i++
            j++
      }
      child.InitVal()
      return &child
}

func randomSegment() (int, int) {
      start := rand.Intn(125)
```

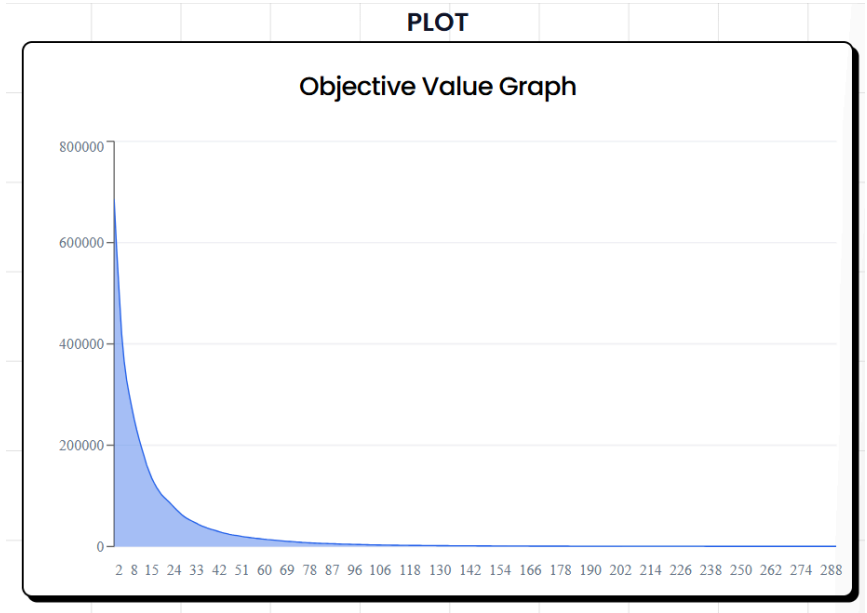
```
	end := rand.Intn(125)
	for start == end {
		end = rand.Intn(125)
	}
	return start, end
}

func mutation(state *models.State) {
	first := rand.Intn(125)
	second := rand.Intn(125)
	segLen := (second - first + 1 + 125) % 125

	rot := 0
	if segLen != 0 {
		rot = rand.Intn(segLen)
	}

	temp := make([]int, segLen)
	for i := 0; i < segLen; i++ {
		srcIdx := (first + i) % 125
		targetIdx := (i + rot) % segLen
		temp[targetIdx] = state.Cubes[srcIdx]
	}

	for i := 0; i < segLen; i++ {
		state.Cubes[(first+i)%125] = temp[i]
	}
}
```

BAB IV
HASIL EKSPERIMEN

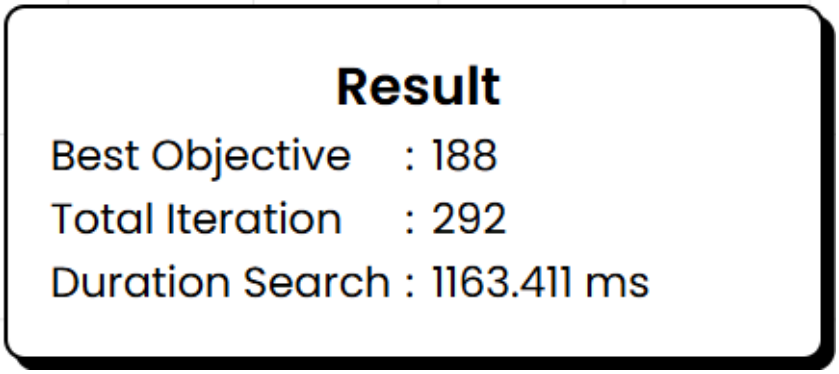
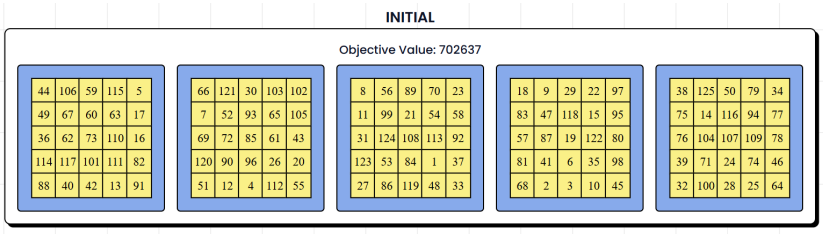
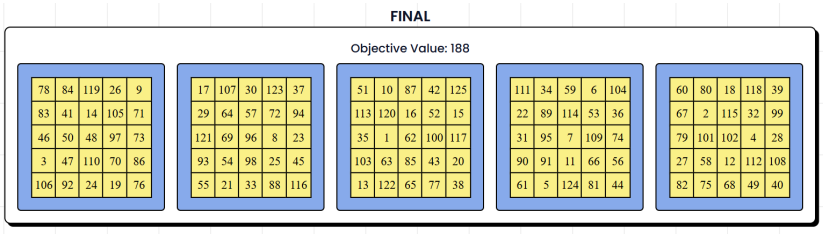
A. Steepest Ascent Hill-Climbing

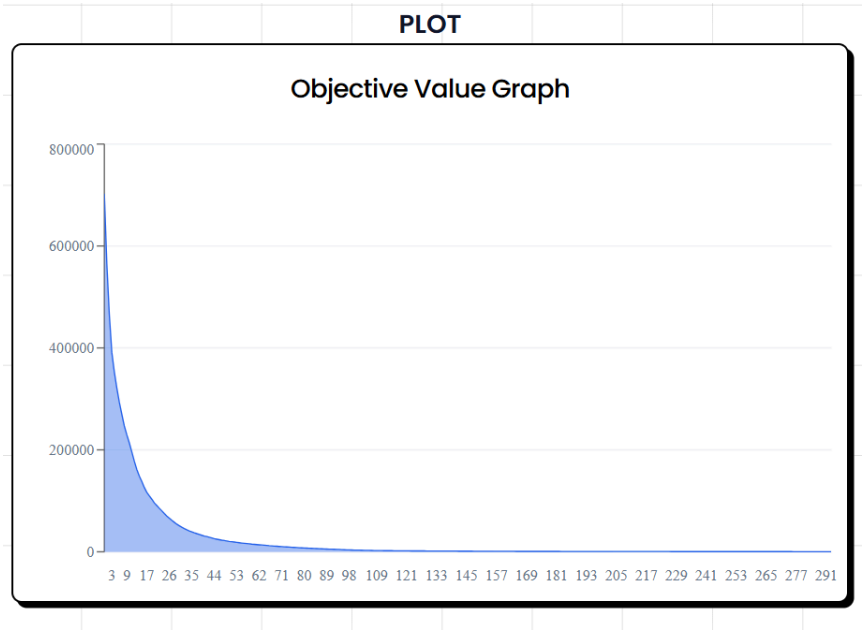
Percobaan 1

Objective Value dan Durasi	**Result** Best Objective : 261 Total Iteration : 289 Duration Search : 1158.205 ms
State Awal	INITIAL Objective Value: 685996 48 51 89 75 53 / 125 91 65 59 80 / 109 2 19 35 93 / 16 78 55 111 72 / 115 22 113 69 7 103 96 106 122 14 / 68 12 121 27 114 / 61 15 13 52 29 / 74 104 124 42 31 / 9 116 56 36 92 73 87 79 21 40 / 66 107 99 46 77 / 28 119 20 58 24 / 41 26 63 57 110 / 94 17 86 102 33 101 76 98 120 84 / 64 108 123 50 25 / 6 118 45 34 18 / 62 1 95 82 44 / 5 4 54 32 30 90 67 37 100 105 / 71 43 112 10 81 / 3 49 47 88 85 / 60 23 83 97 38 / 70 39 8 117 11
State Akhir	FINAL Objective Value: 261 42 47 110 85 33 / 70 99 14 56 77 / 125 13 64 28 82 / 37 123 7 68 80 / 40 34 120 78 43 11 97 3 119 84 / 31 46 107 35 96 / 121 9 103 81 2 / 100 41 92 53 29 / 51 122 12 24 104 71 106 74 39 25 / 66 75 54 114 6 / 23 109 62 45 76 / 60 20 98 17 118 / 94 4 27 102 90 101 1 113 15 86 / 61 63 22 89 79 / 26 69 49 67 105 / 108 72 5 93 38 / 19 112 124 52 8 91 65 16 55 87 / 88 32 117 21 57 / 18 116 36 95 50 / 10 59 115 83 48 / 111 44 30 58 73

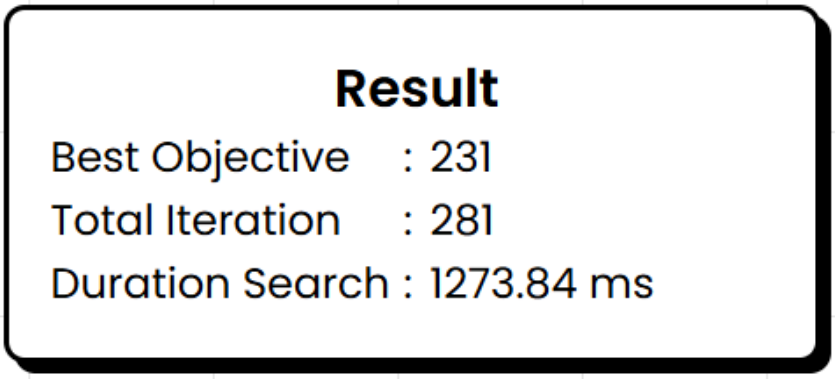
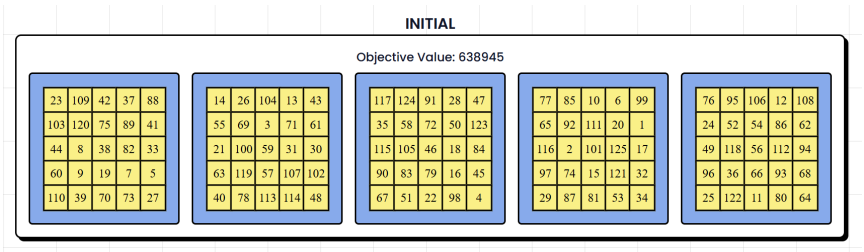
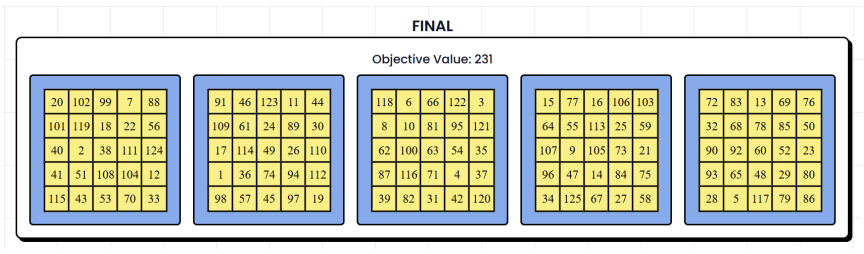
Plot Objective Value terhadap Iterasi	

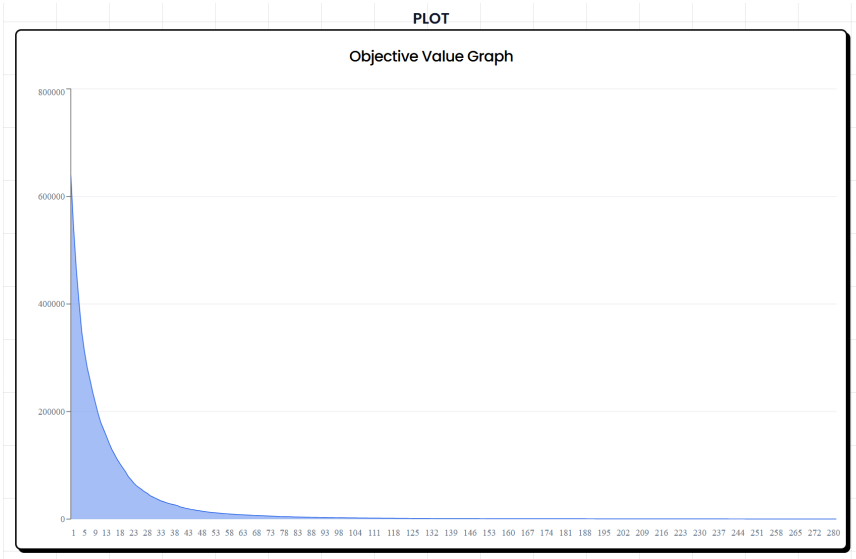
Percobaan 2

Objective Value dan Durasi	
State Awal	
State Akhir	

Plot Objective Value terhadap Iterasi	

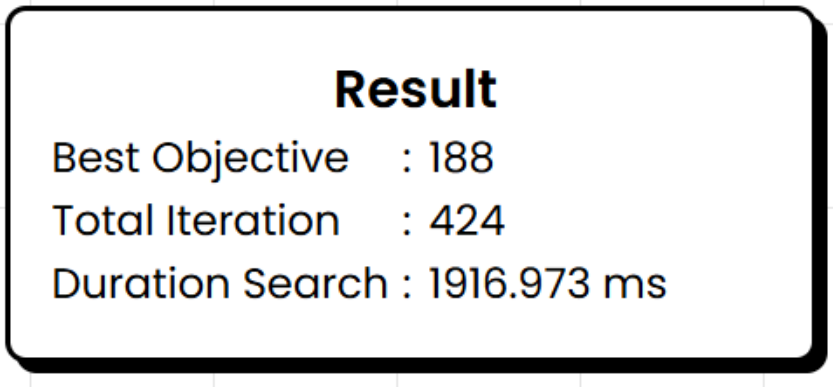
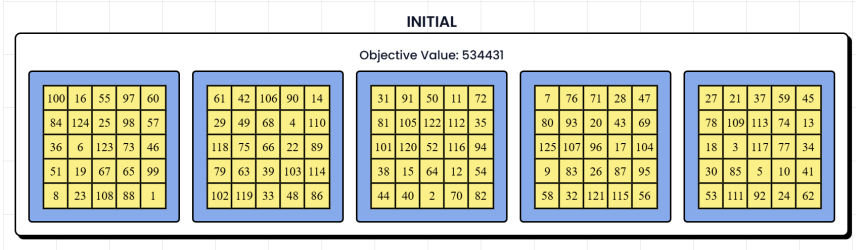
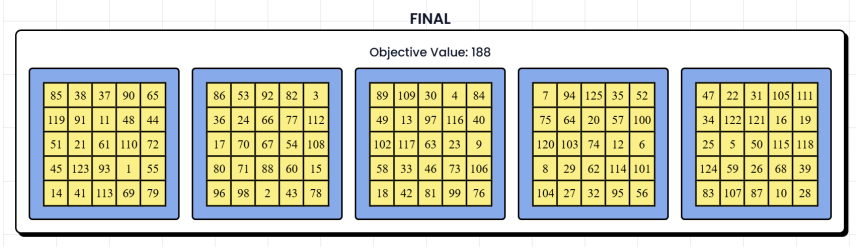
Percobaan 3

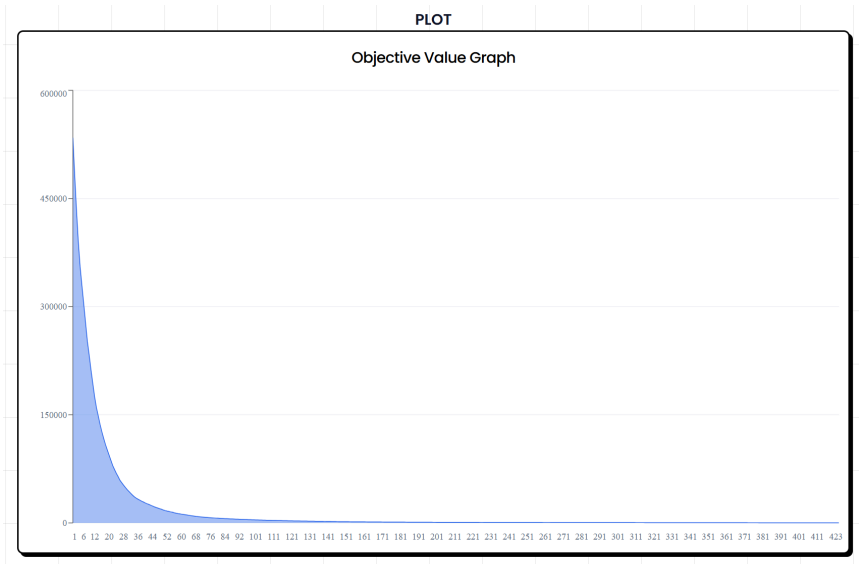
Objective Value dan Durasi	
State Awal	
State Akhir	

Plot Objective Value terhadap Iterasi	

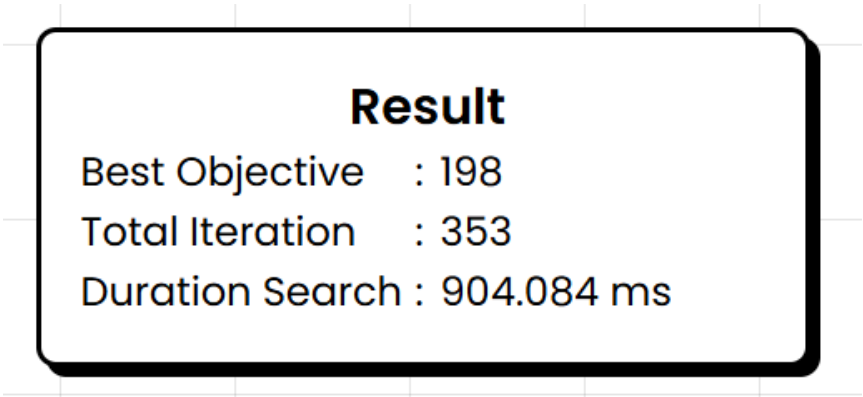
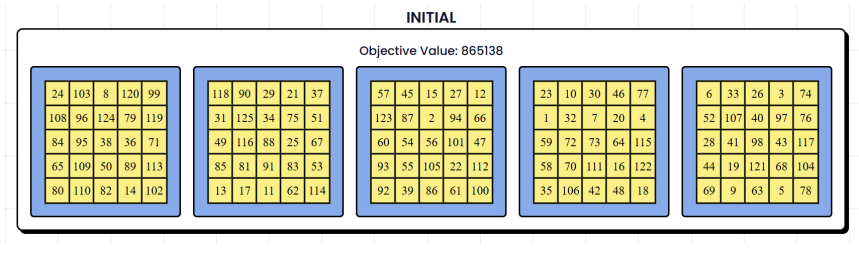
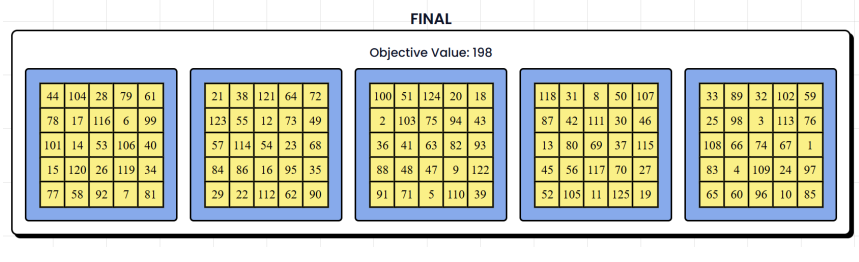
B. Hill-Climbing with Sideways Move

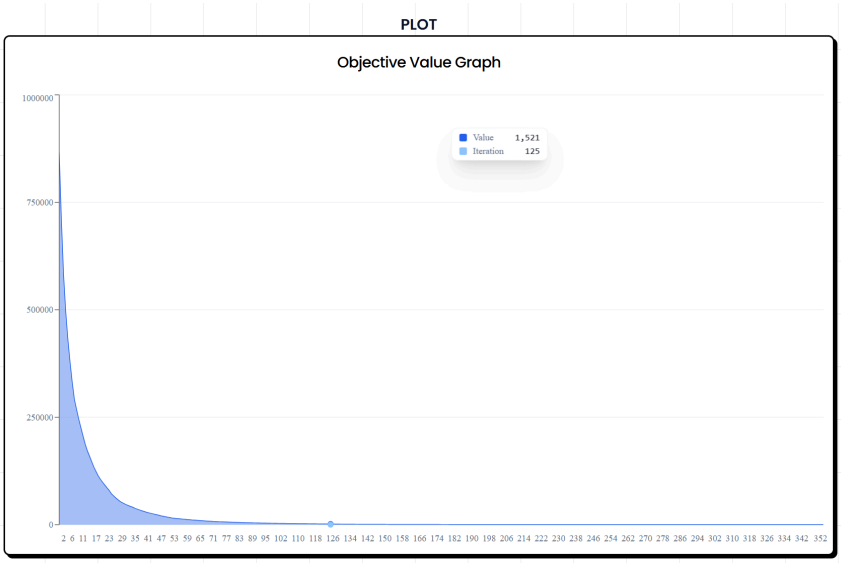
Percobaan 1: Max Iteration 10

Objective Value dan Durasi	
State Awal	
State Akhir	

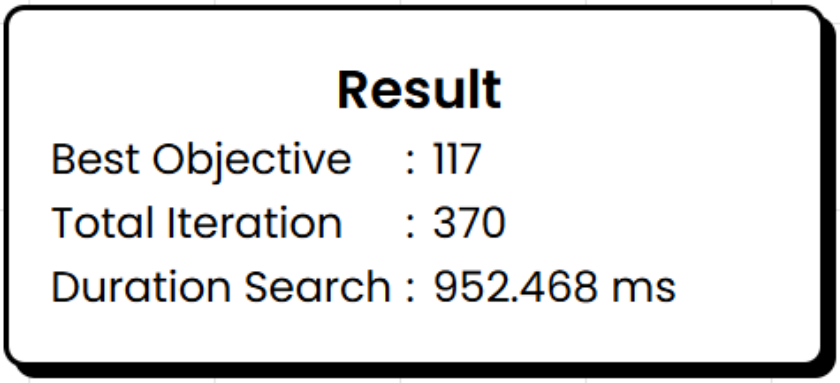
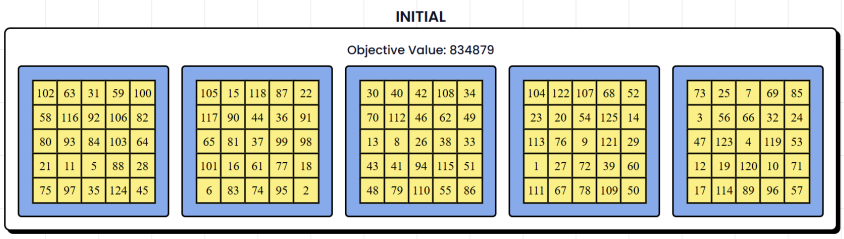
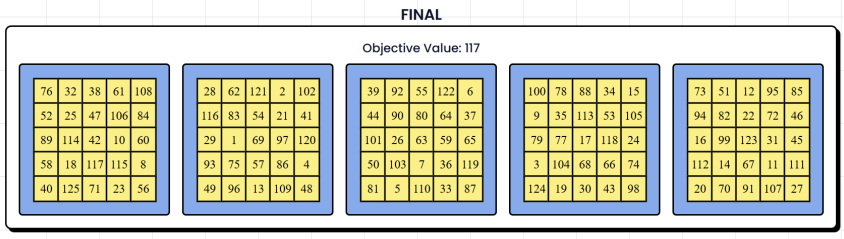
Plot Objective Value terhadap Iterasi	

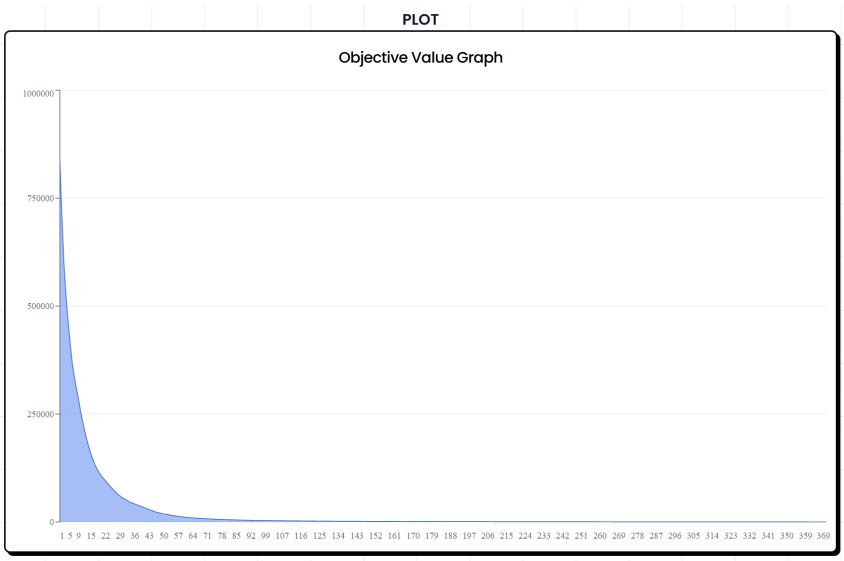
Percobaan 2: Max Iteration 100

Objective Value dan Durasi	
State Awal	
State Akhir	

Plot Objective Value terhadap Iterasi	

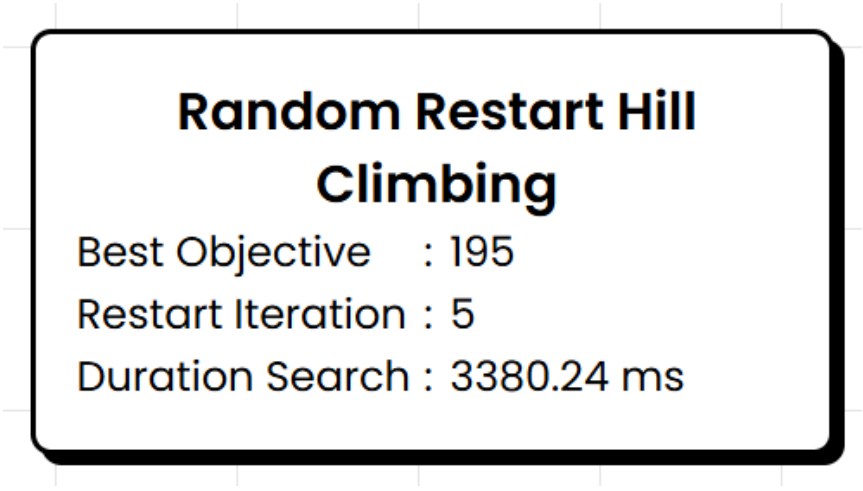
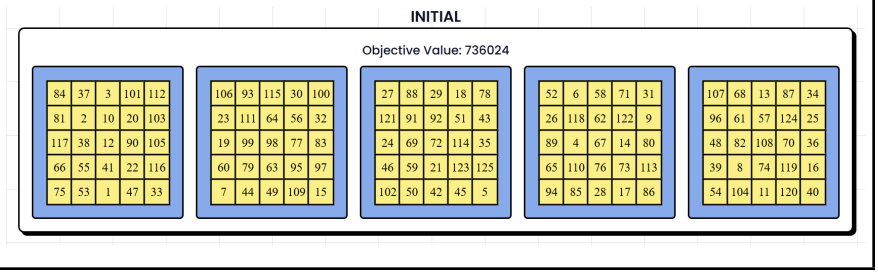
Percobaan 3: Max Iteration 1000

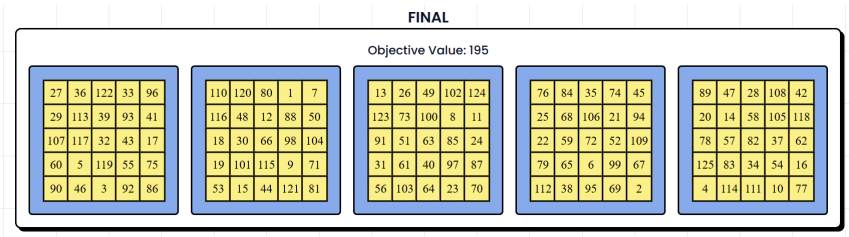
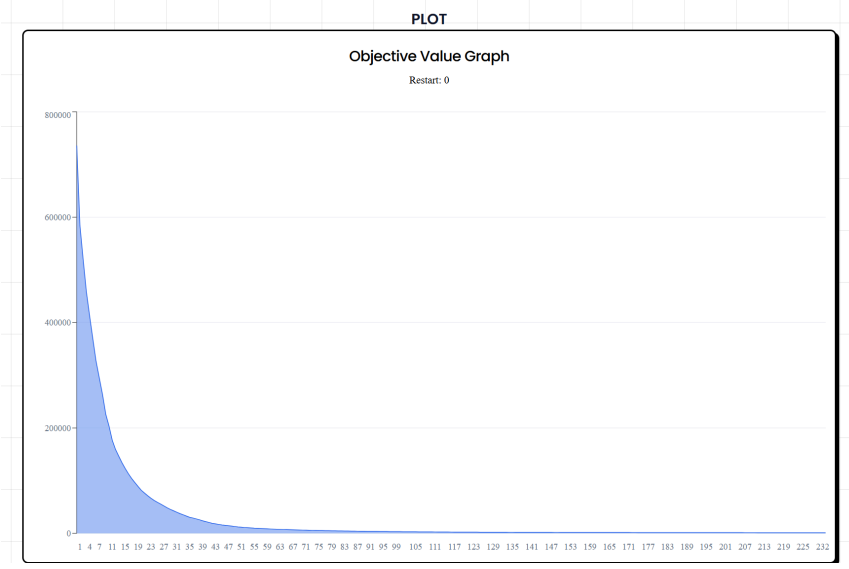
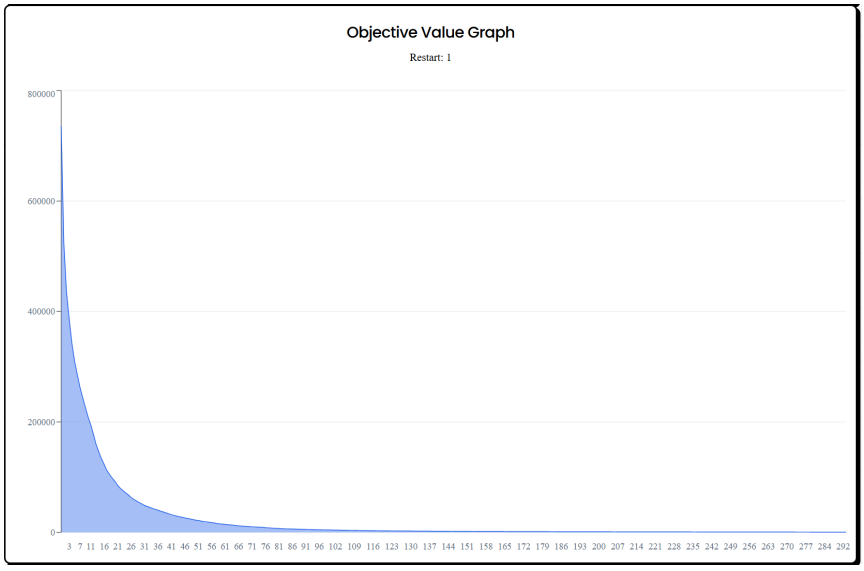
Objective Value dan Durasi	
State Awal	
State Akhir	

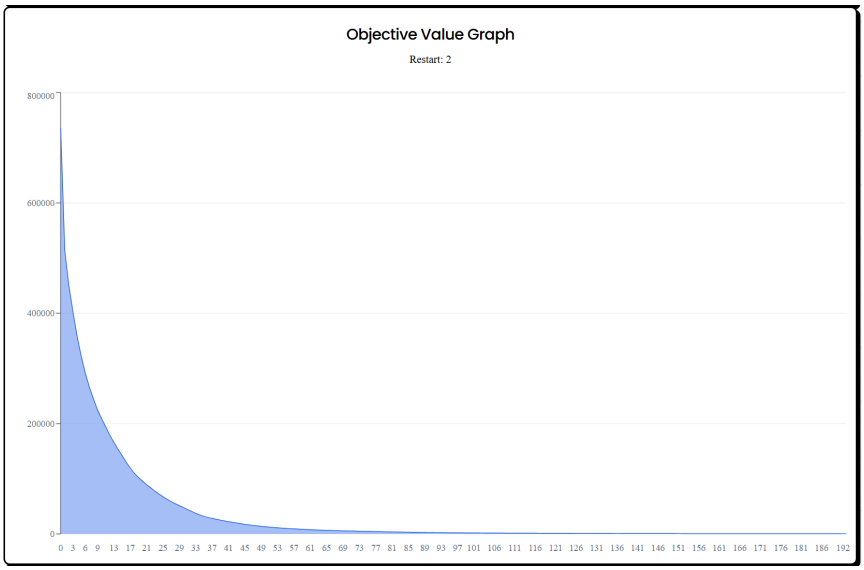
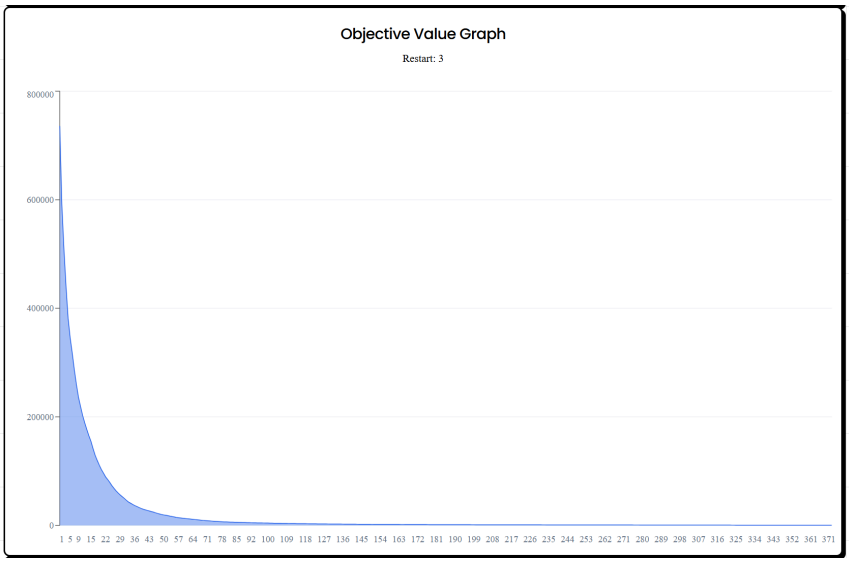
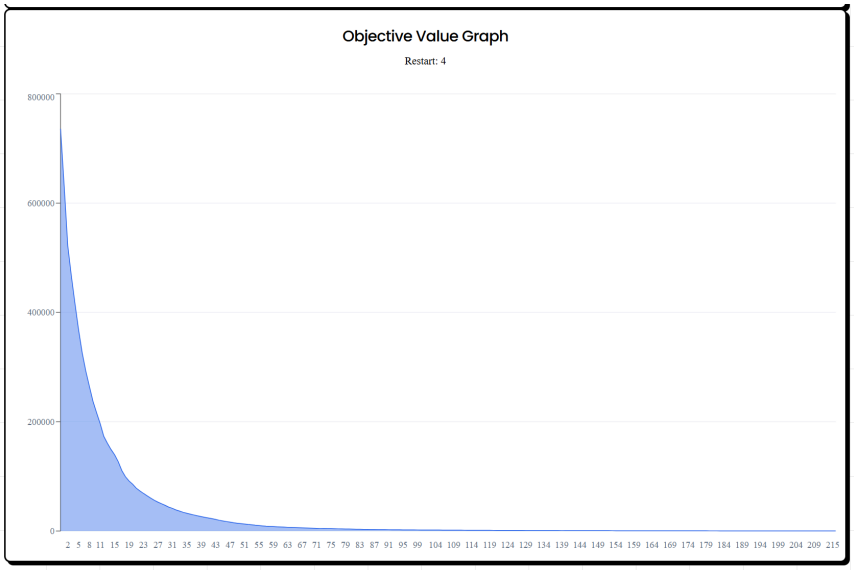
Plot Objective Value terhadap Iterasi	

C. Hill-Climbing with Random Restart

Percobaan 1: 5 Restart

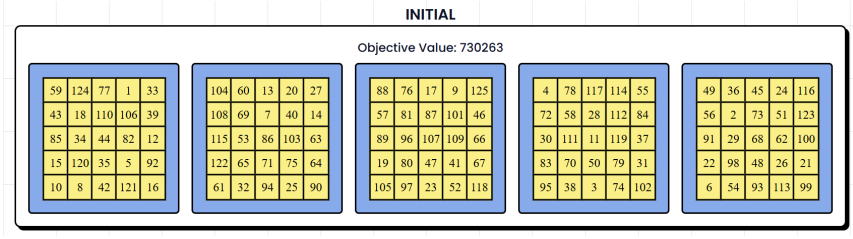
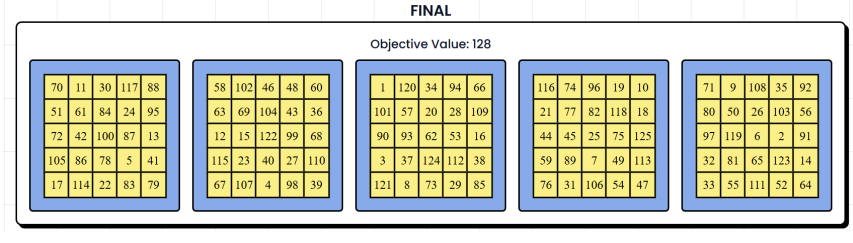
Objective Value dan Durasi	
State Awal Iterasi	

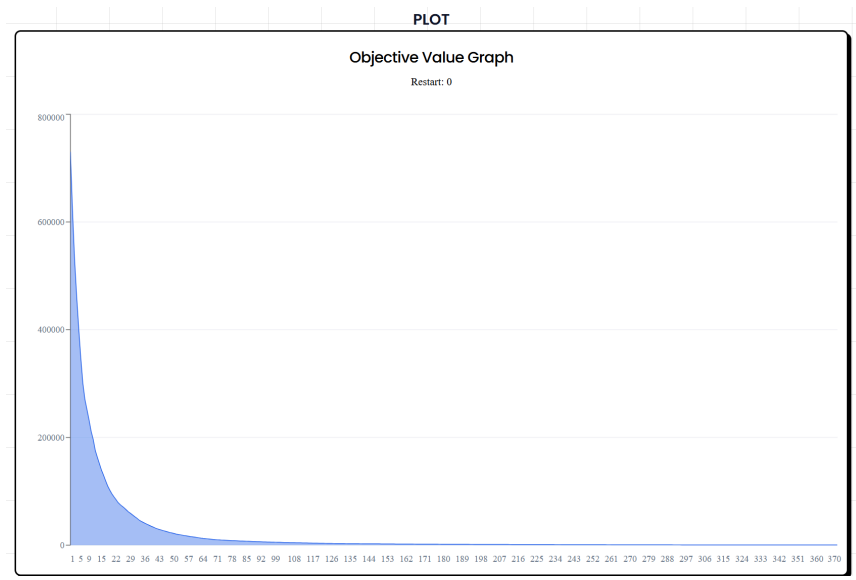
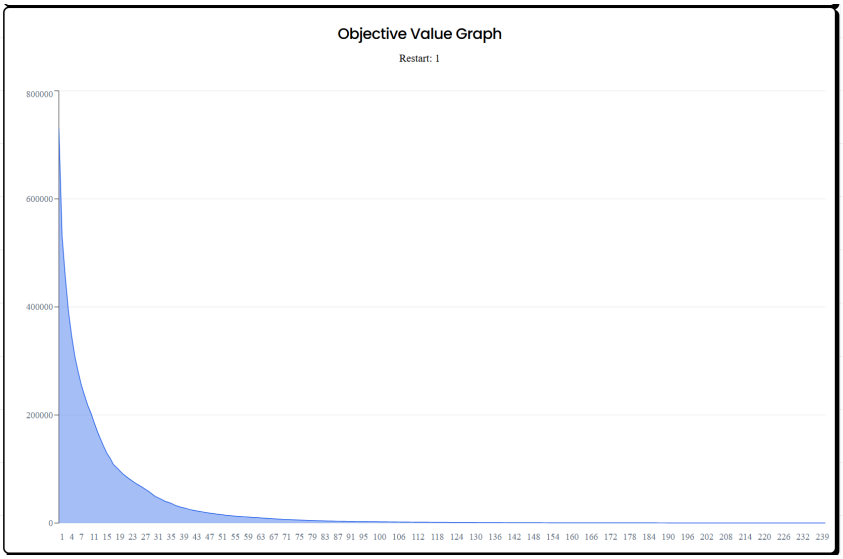
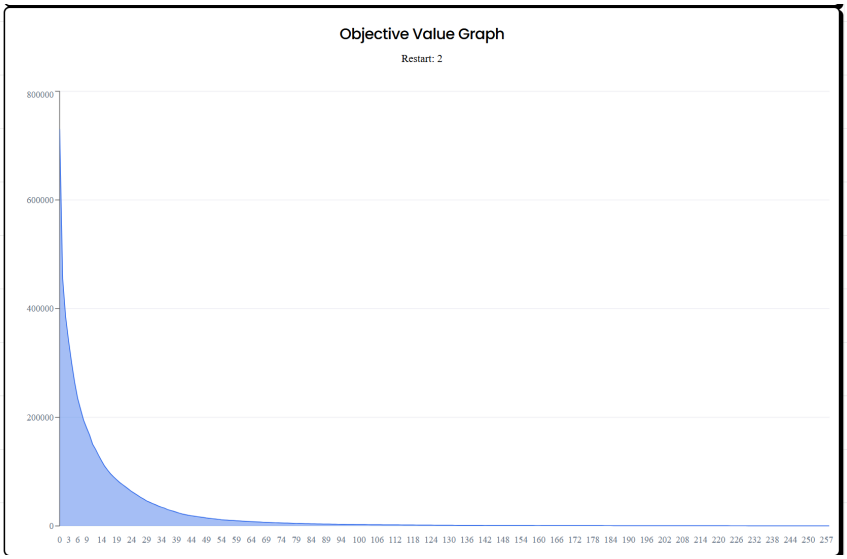
State Akhir Iterasi	
Plot Objective Value terhadap Iterasi 1	
Plot Objective Value terhadap Iterasi 2	

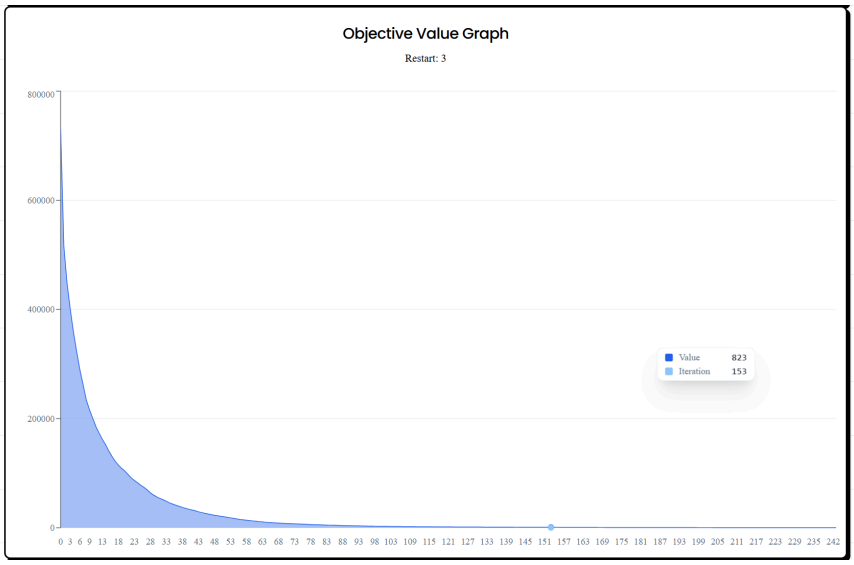
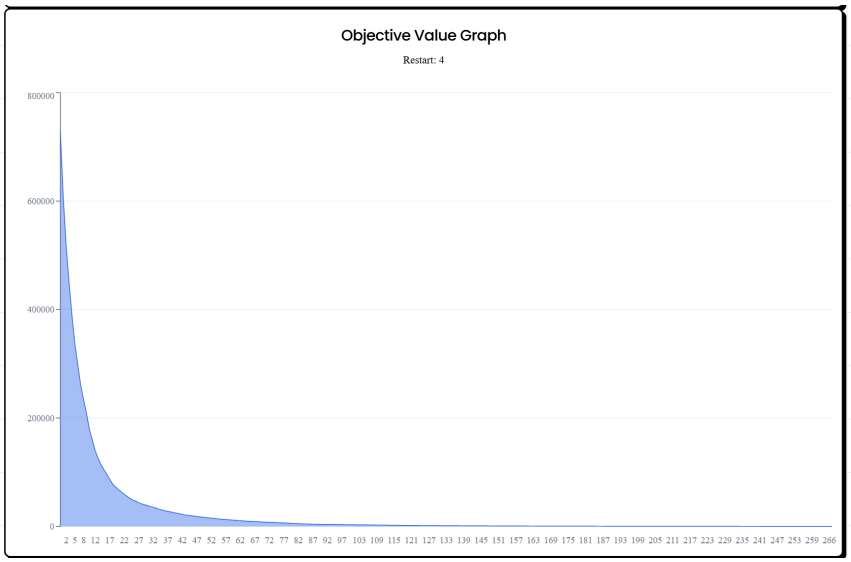
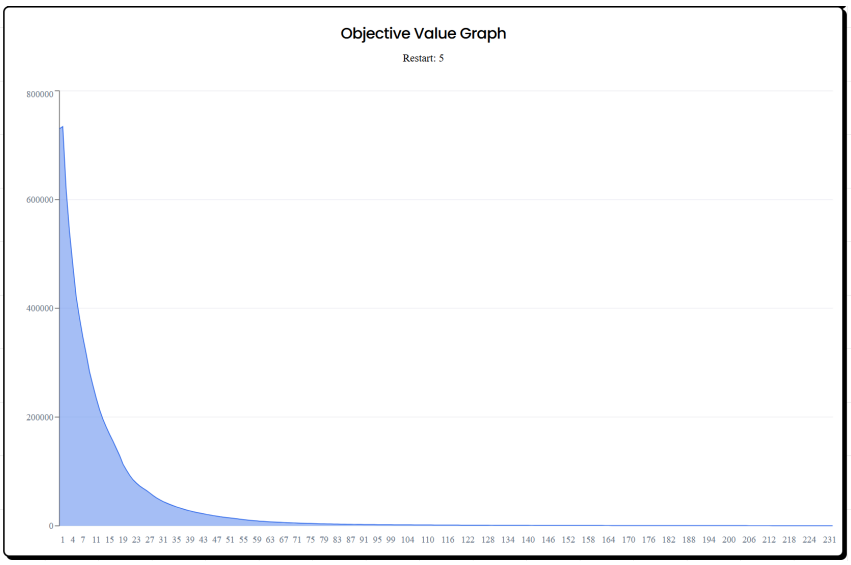
Plot Objective Value terhadap Iterasi 3	
Plot Objective Value terhadap Iterasi 4	
Plot Objective Value terhadap Iterasi 5	

Iteration untuk setiap restart	Iteration for restart 0 : 233 Iteration for restart 1 : 293 Iteration for restart 2 : 193 Iteration for restart 3 : 372 Iteration for restart 4 : 216

Percobaan 2: 6 Restart

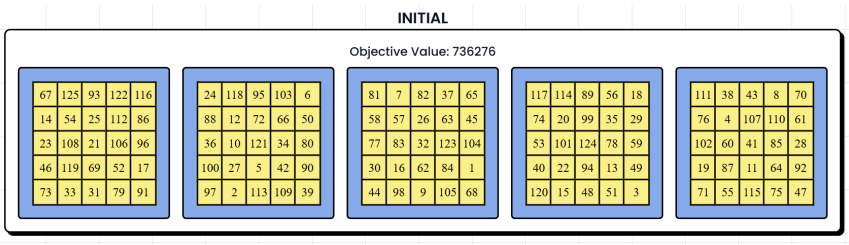
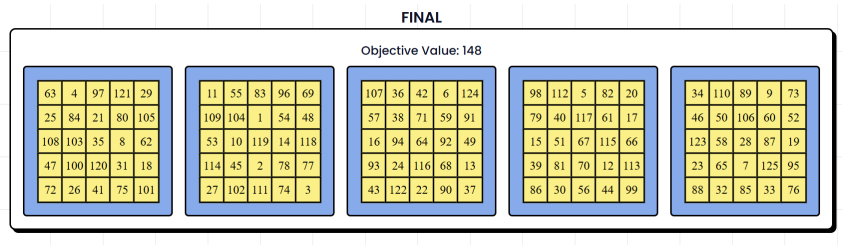
Objective Value dan Durasi	**Random Restart Hill Climbing** Best Objective : 128 Restart Iteration : 6 Duration Search : 4197.566 ms
State Awal Iterasi	
State Akhir Iterasi	

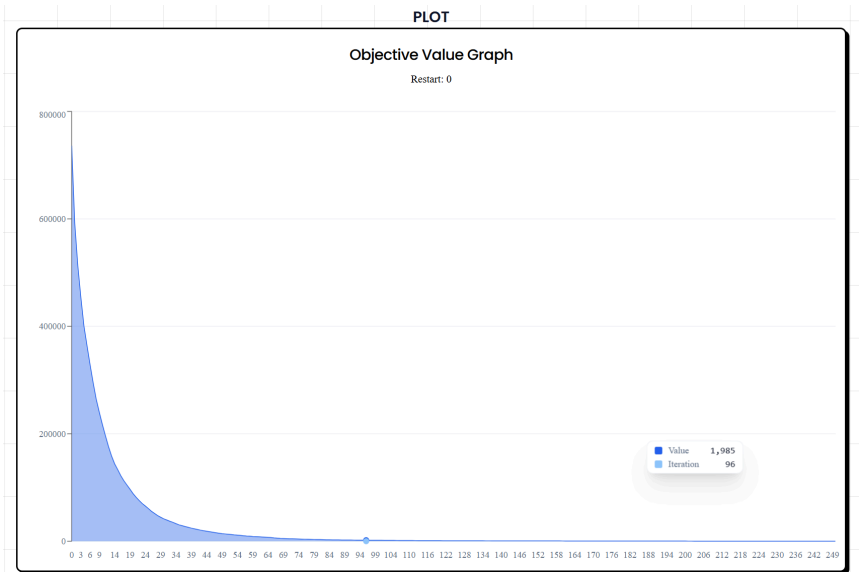
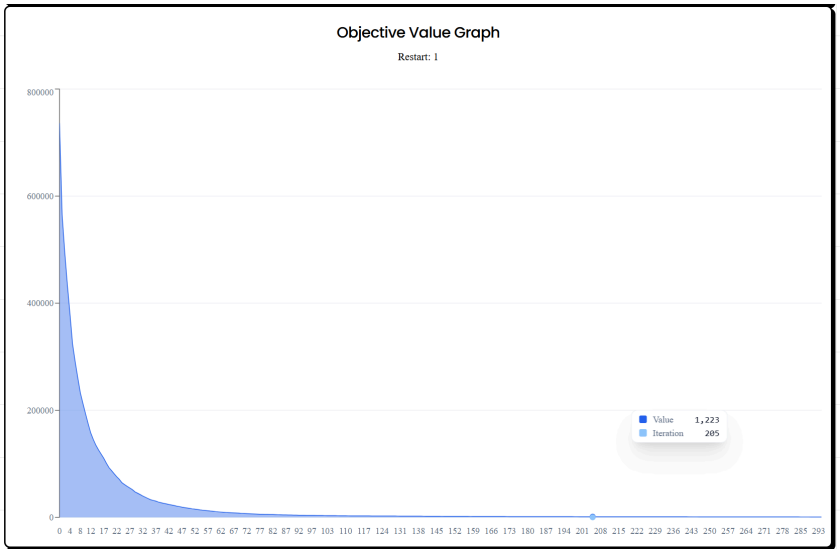
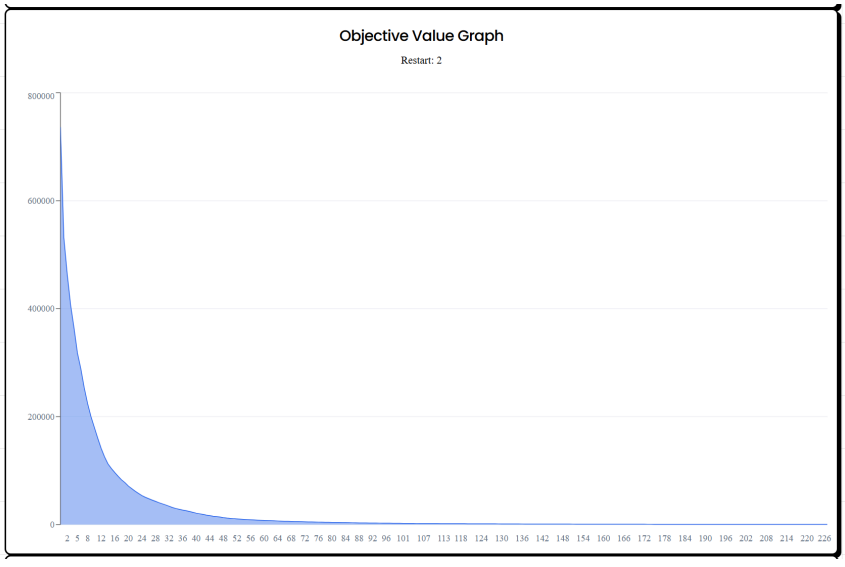
Plot Objective Value terhadap Iterasi 1	
Plot Objective Value terhadap Iterasi 2	
Plot Objective Value terhadap Iterasi 3	

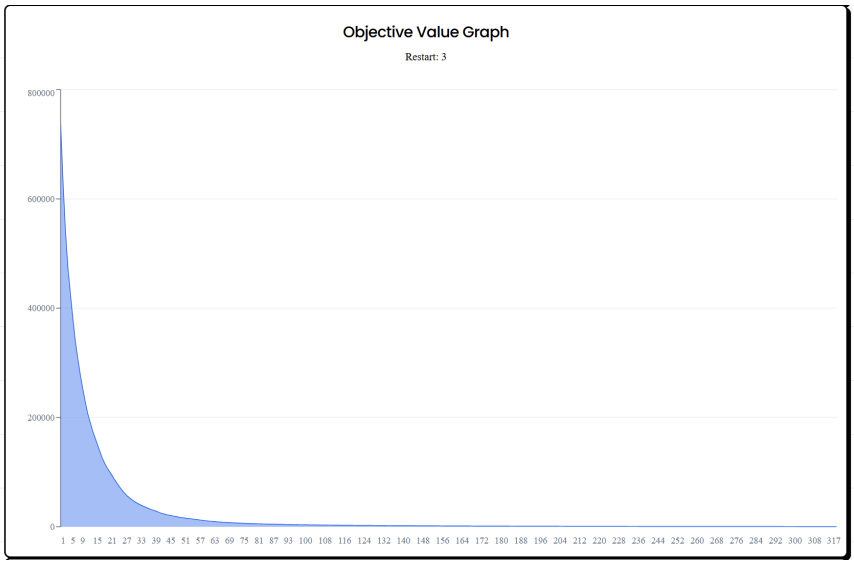
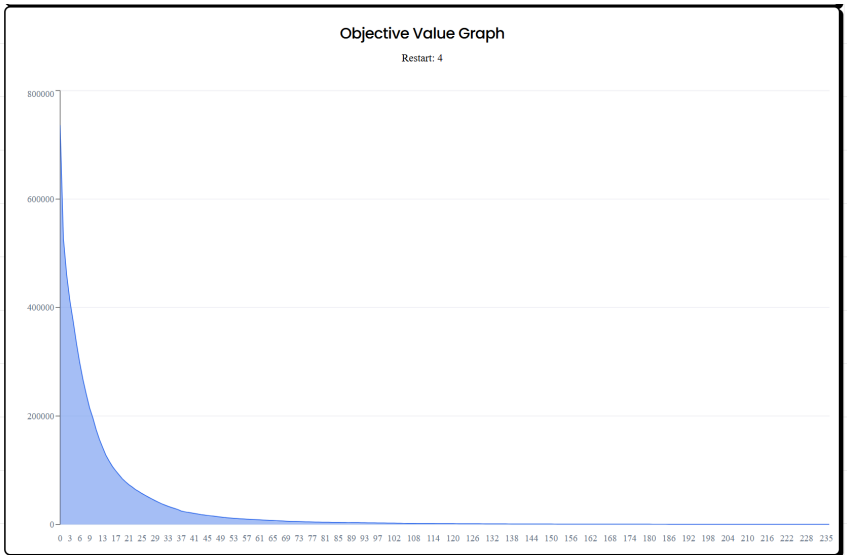
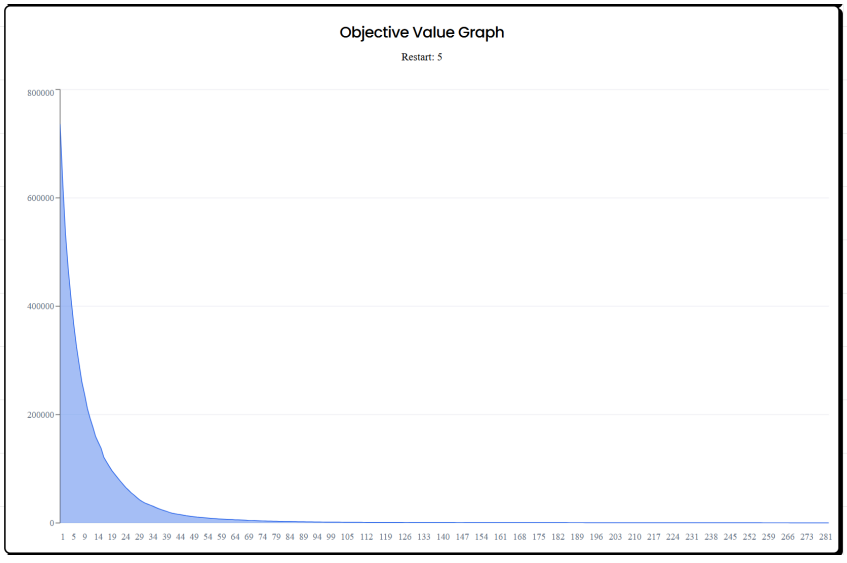
Plot Objective Value terhadap Iterasi 4	
Plot Objective Value terhadap Iterasi 5	
Plot Objective Value terhadap Iterasi 6	

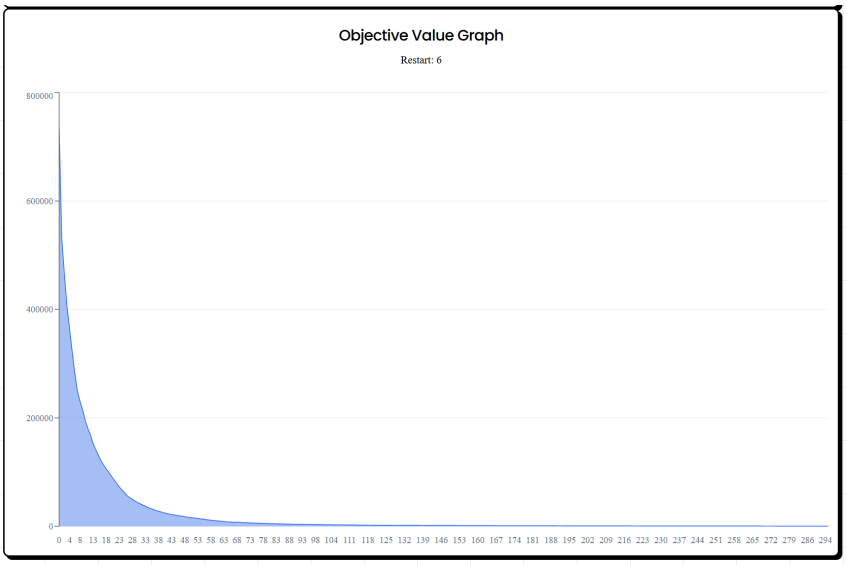
Iteration untuk setiap restart	Iteration for restart 0 : 371 Iteration for restart 1 : 240 Iteration for restart 2 : 258 Iteration for restart 3 : 243 Iteration for restart 4 : 267 Iteration for restart 5 : 232

Percobaan 3: 7 Restart

Objective Value dan Durasi	**Random Restart Hill Climbing** Best Objective : 148 Restart Iteration : 7 Duration Search : 4990.691 ms
State Awal Iterasi	
State Akhir Iterasi	

Plot Objective Value terhadap Iterasi 1	
Plot Objective Value terhadap Iterasi 2	
Plot Objective Value terhadap Iterasi 3	

Plot Objective Value terhadap Iterasi 4	
Plot Objective Value terhadap Iterasi 5	
Plot Objective Value terhadap Iterasi 5	

Plot Objective Value terhadap Iterasi 6	
Iteration untuk setiap restart	Iteration for restart 0 : 250 Iteration for restart 1 : 294 Iteration for restart 2 : 227 Iteration for restart 3 : 318 Iteration for restart 4 : 236 Iteration for restart 5 : 282 Iteration for restart 6 : 295

D. Stochastic Hill-Climbing

Percobaan 1

Objective Value dan Durasi	**Result** Best Objective : 229 Total Iteration : 800000 Duration Search : 1118.706 ms
State Awal	**INITIAL** Objective Value: 694845 38 88 106 104 16 / 4 113 22 69 124 / 11 65 115 114 56 / 64 1 10 17 63 / 111 59 28 37 58 118 77 55 35 54 / 95 9 2 34 42 / 81 83 80 23 61 / 46 87 32 72 21 / 103 75 101 15 85 51 48 89 36 110 / 68 116 8 123 45 / 79 121 29 47 66 / 99 52 120 27 33 / 94 40 7 96 30 112 57 108 102 12 / 105 93 31 49 109 / 117 122 50 25 62 / 90 82 70 73 67 / 19 41 3 18 100 98 53 71 13 20 / 92 76 125 26 97 / 107 24 78 14 91 / 86 44 74 39 119 / 84 5 43 6 60
State Akhir	**FINAL** Objective Value: 229 52 83 113 40 26 / 106 16 13 80 100 / 67 56 87 55 50 / 1 31 94 107 85 / 90 125 9 33 54 28 112 29 103 44 / 10 93 59 45 109 / 98 8 64 95 49 / 86 69 39 68 53 / 92 35 123 4 61 78 47 60 21 110 / 75 122 18 88 11 / 34 38 62 63 117 / 114 43 82 24 51 / 15 65 91 118 27 71 42 7 74 119 / 20 48 101 76 72 / 102 96 36 79 3 / 111 73 81 46 5 / 12 57 89 41 115 84 32 105 77 17 / 104 37 124 25 23 / 14 116 66 22 97 / 6 99 19 70 121 / 108 30 2 120 58
Plot Objective Value terhadap Iterasi	**PLOT** Objective Value Graph 800000 600000 400000 200000 0 14 33 52 71 90 111 136 161 186 211 236 261 286 311 336 361 386 411 436 461 486 511 536 561 586 615

Percobaan 2

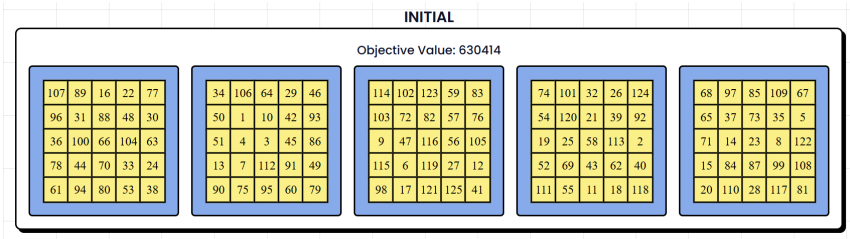
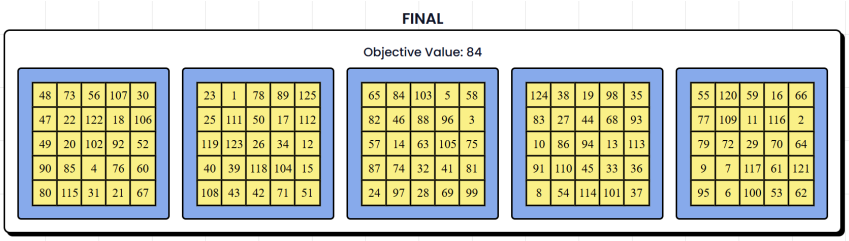
Objective Value dan Durasi	**Result** Best Objective : 205 Total Iteration : 800000 Duration Search : 1122.417 ms
State Awal	**INITIAL** Objective Value: 613775 110 26 108 53 36 / 31 77 111 7 88 / 50 97 51 55 5 / 104 29 14 90 52 / 11 57 96 81 93 49 125 67 69 84 / 71 39 17 34 100 / 117 43 99 124 32 / 30 45 101 107 95 / 116 62 119 42 87 64 21 94 54 65 / 56 47 70 28 18 / 61 102 113 9 79 / 16 85 66 83 38 / 6 121 27 63 74 91 72 15 76 58 / 33 12 37 48 73 / 120 22 2 114 10 / 3 20 78 106 118 / 19 112 23 24 122 59 13 103 68 1 / 89 92 25 105 123 / 82 75 8 80 44 / 60 86 40 115 41 / 35 98 109 46 4
State Akhir	**FINAL** Objective Value: 205 87 28 34 119 48 / 29 59 114 4 109 / 10 123 90 71 22 / 92 78 62 12 70 / 97 26 18 107 66 17 112 99 64 24 / 79 60 11 55 111 / 80 15 103 101 16 / 49 43 8 93 122 / 91 86 95 3 41 54 19 67 94 83 / 98 47 77 88 6 / 37 125 63 14 72 / 121 76 65 25 30 / 5 46 42 96 124 106 84 1 7 117 / 82 45 113 68 9 / 69 13 38 108 85 / 23 58 105 75 53 / 35 116 57 56 51 50 73 115 33 44 / 27 104 2 100 81 / 118 36 21 20 120 / 31 61 74 110 39 / 89 40 102 52 32
Plot Objective Value terhadap Iterasi	**PLOT** Objective Value Graph 800000 600000 400000 200000 0 13 34 55 76 97 121 149 177 205 233 261 289 317 345 373 401 429 457 485 513 541 569 597 625 653 686

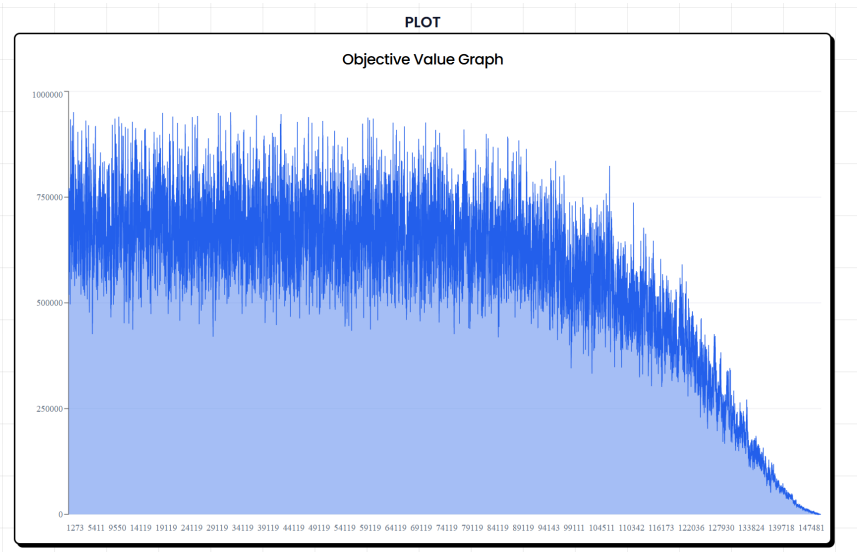
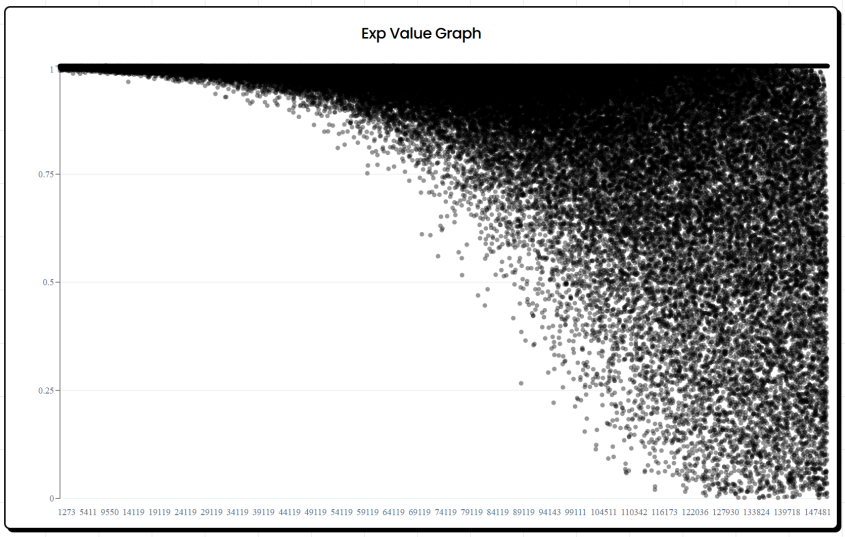
Percobaan 3

Objective Value dan Durasi	**Result** Best Objective : 302 Total Iteration : 800000 Duration Search : 1125.872 ms
State Awal	**INITIAL** Objective Value: 618070 45 99 79 58 109 / 89 70 87 9 2 / 26 57 29 95 40 / 105 69 78 43 15 / 38 56 34 62 48 61 32 64 121 115 / 8 97 18 23 81 / 112 82 66 1 19 / 14 55 30 85 106 / 124 41 63 71 50 123 20 60 13 37 / 90 36 72 119 46 / 75 101 76 52 22 / 93 33 74 51 98 / 49 53 12 25 42 103 21 108 118 111 / 122 80 24 47 114 / 44 4 17 39 35 / 67 73 16 84 7 / 100 77 113 120 28 68 116 6 107 117 / 10 94 11 31 104 / 5 3 88 54 91 / 65 102 96 125 86 / 92 59 110 27 83
State Akhir	**FINAL** Objective Value: 535 78 62 15 71 89 / 87 102 51 45 28 / 18 93 26 125 50 / 42 57 98 39 80 / 88 1 124 33 70 19 65 48 113 69 / 25 66 118 23 85 / 109 106 96 2 4 / 56 47 20 83 108 / 105 30 34 97 49 122 43 119 24 8 / 7 5 86 101 117 / 58 36 38 116 67 / 114 112 44 35 10 / 13 120 27 40 115 74 32 91 11 107 / 73 64 54 55 68 / 9 63 79 61 103 / 100 72 53 75 14 / 60 84 37 111 22 21 110 41 95 46 / 123 77 6 94 17 / 121 16 76 12 90 / 3 29 99 82 104 / 52 81 92 31 59
Plot Objective Value terhadap Iterasi	**PLOT** Objective Value Graph 1000000 750000 500000 250000 0 11 31 52 73 94 118 145 173 201 229 257 285 313 341 369 397 425 453 481 509 537 565 593 621 649 681

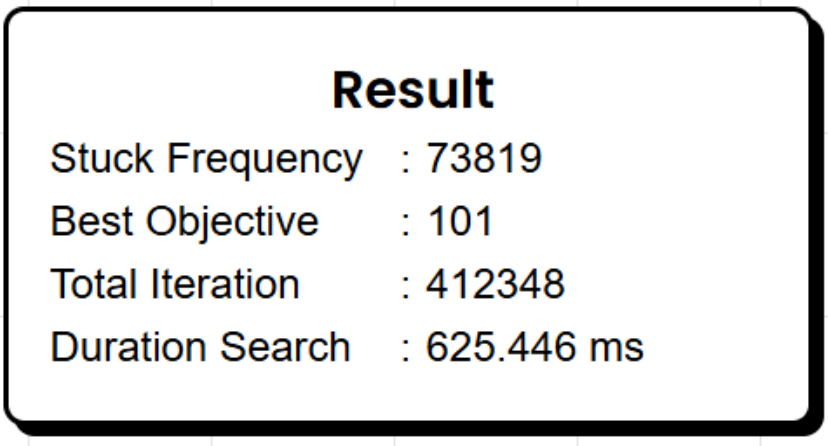
E. Simulated Annealing

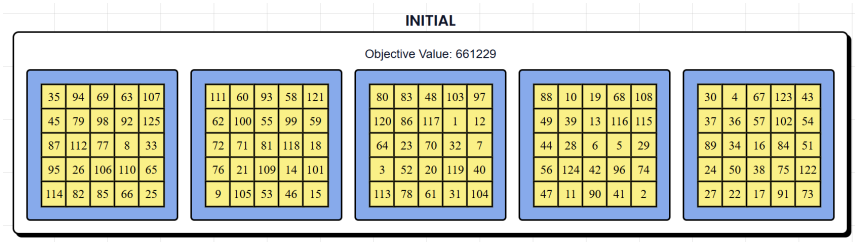
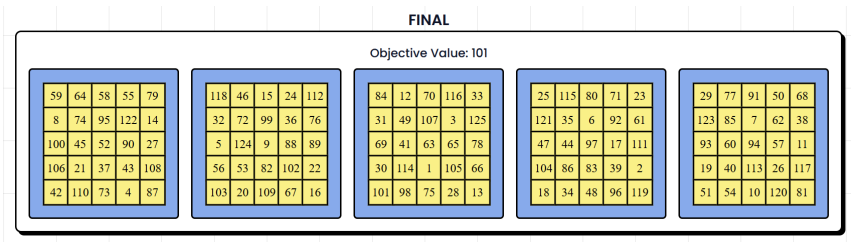
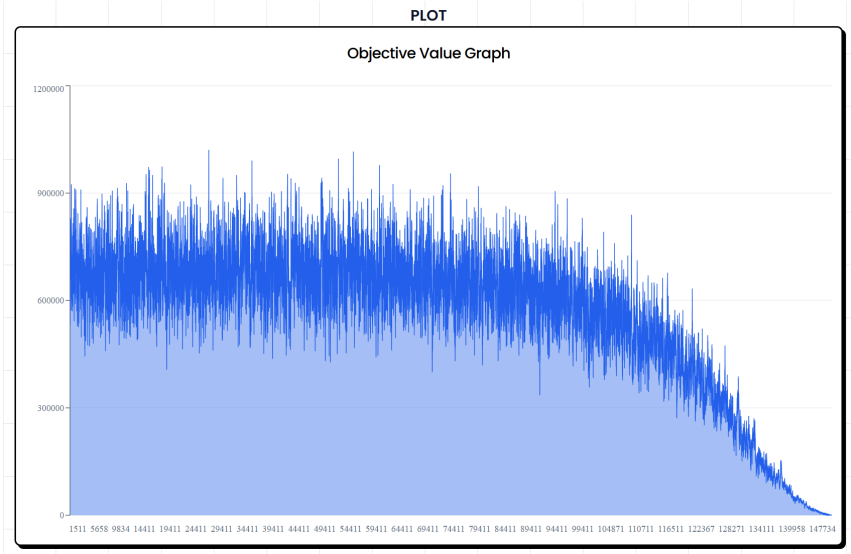
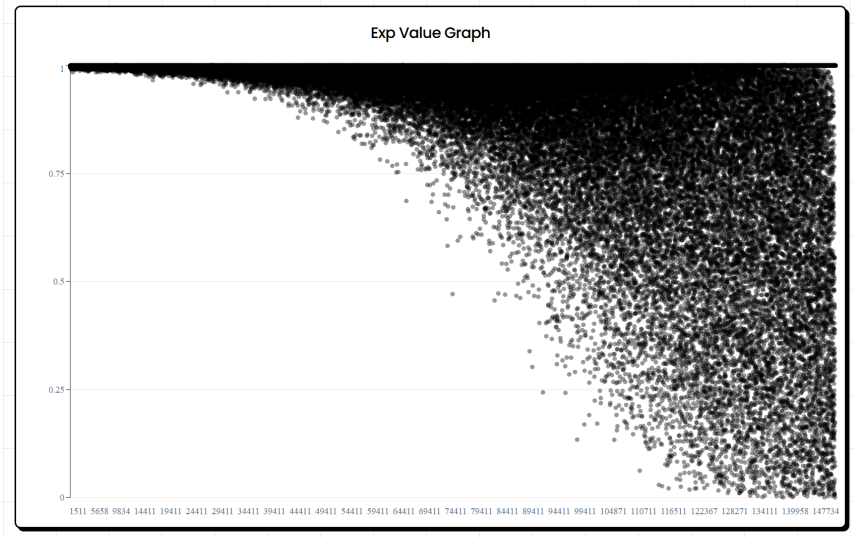
Percobaan 1

Objective Value dan Durasi	**Result** Stuck Frequency : 73836 Best Objective : 84 Total Iteration : 412348 Duration Search : 622.29 ms
State Awal	
State Akhir	

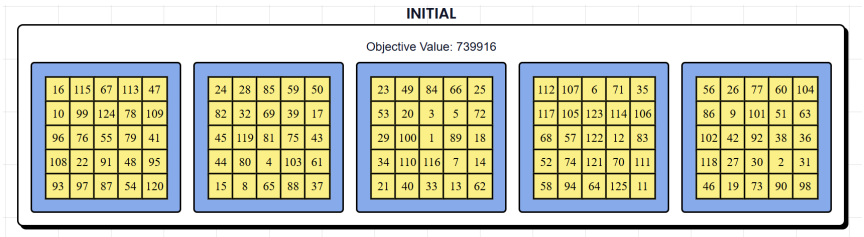
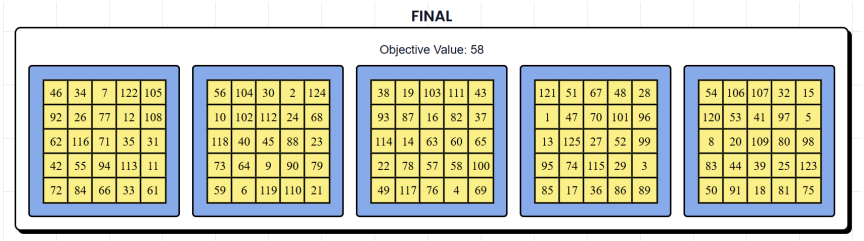
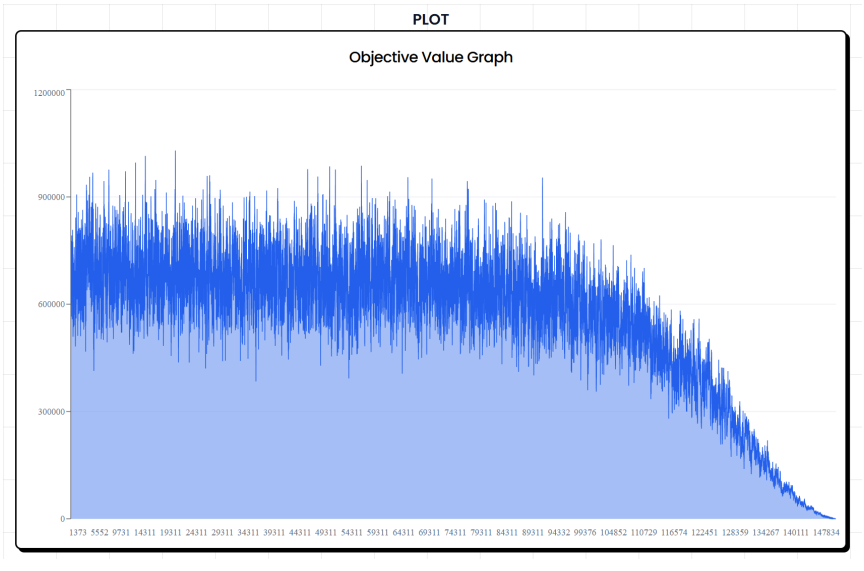
Plot Objective Value terhadap Iterasi	
Plot $e^{\Delta E/T}$ terhadap iterasi	

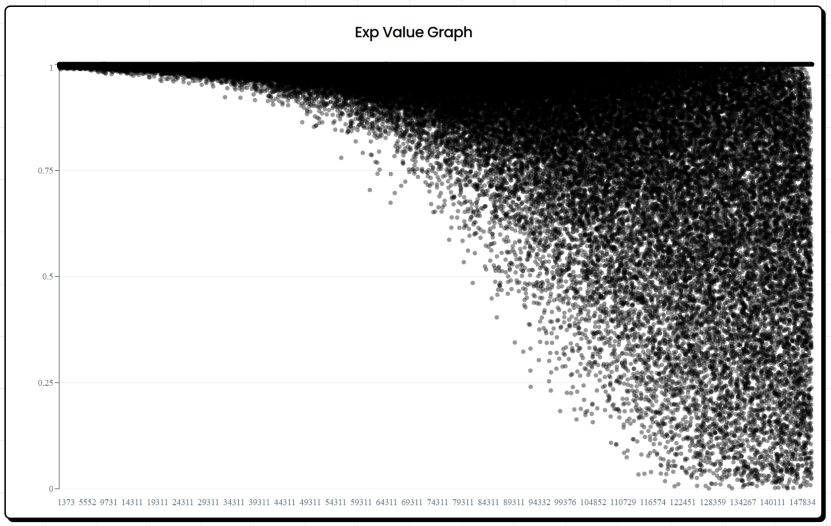
Percobaan 2

Objective Value dan Durasi	

State Awal	
State Akhir	
Plot Objective Value terhadap Iterasi	
Plot $e^{\Delta E/T}$ terhadap iterasi	

Percobaan 3

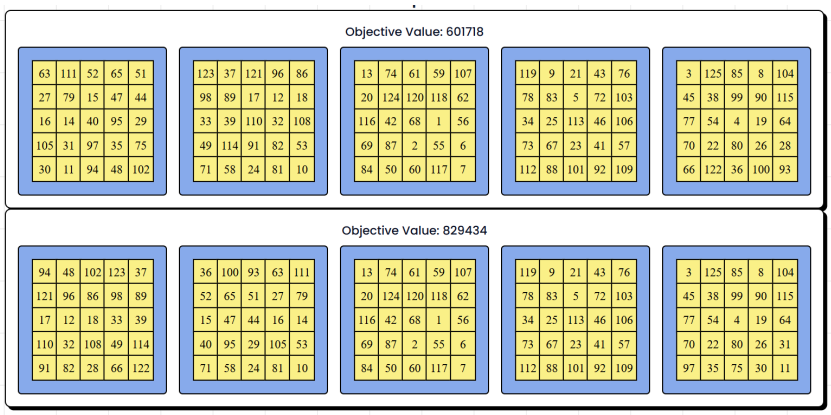
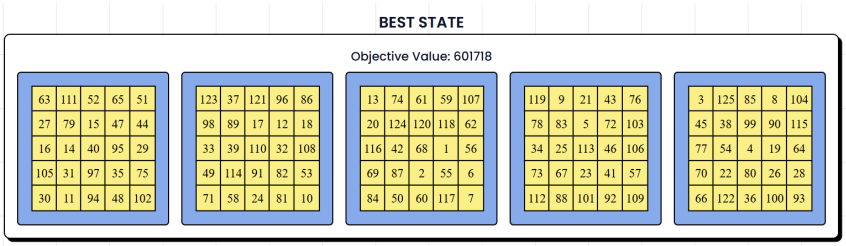
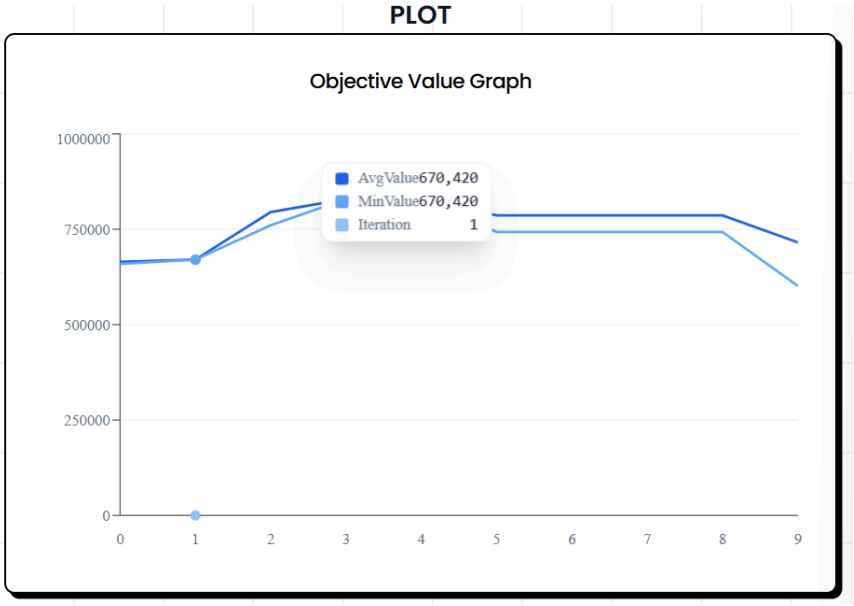
Objective Value dan Durasi	**Result** Stuck Frequency : 74106 Best Objective : 58 Total Iteration : 412348 Duration Search : 649.611 ms
State Awal	
State Akhir	
Plot Objective Value terhadap Iterasi	

Plot $e^{\Delta E/T}$ terhadap iterasi	

F. Genetic Algorithm

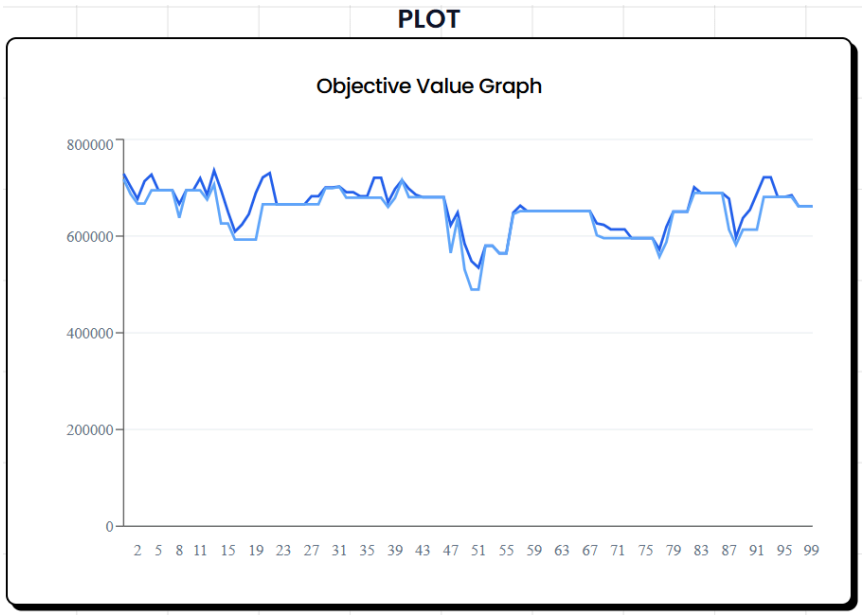
Percobaan 1 : population : 2 & iteration: 10

Objective Value dan Durasi	**Genetic Algorithm** Best Objective : 601718 Total Population : 2 Total Iteration : 10 Duration Search : 0 ms
State Awal	Objective Value: 659207 117 79 11 59 125 63 39 118 10 50 75 123 38 20 19 4 51 68 89 6 73 101 122 23 9 67 98 57 3 58 96 62 21 69 84 106 18 13 36 49 86 7 85 35 95 99 52 27 56 40 97 66 12 47 87 54 112 43 114 90 80 105 121 124 30 31 107 119 78 109 72 29 108 102 46 34 76 45 8 53 25 42 24 83 74 33 115 26 120 55 5 37 94 61 15 103 81 111 77 88 82 32 93 64 110 113 48 17 65 2 44 91 41 28 100 116 1 70 14 22 60 71 104 16 92 Objective Value: 670420 94 48 102 123 37 121 96 86 98 89 17 12 18 33 39 110 32 108 49 114 91 82 28 66 122 36 100 93 63 111 52 65 51 27 79 15 47 44 16 14 40 95 29 105 53 71 58 24 81 10 13 74 61 59 107 20 124 120 118 62 116 42 68 1 56 69 87 2 115 77 54 4 19 64 70 22 80 26 31 97 35 75 30 11 55 6 84 50 60 117 7 119 9 21 43 76 78 83 5 72 103 34 25 113 46 106 73 67 23 41 57 112 88 101 92 109 3 125 85 8 104 45 38 99 90

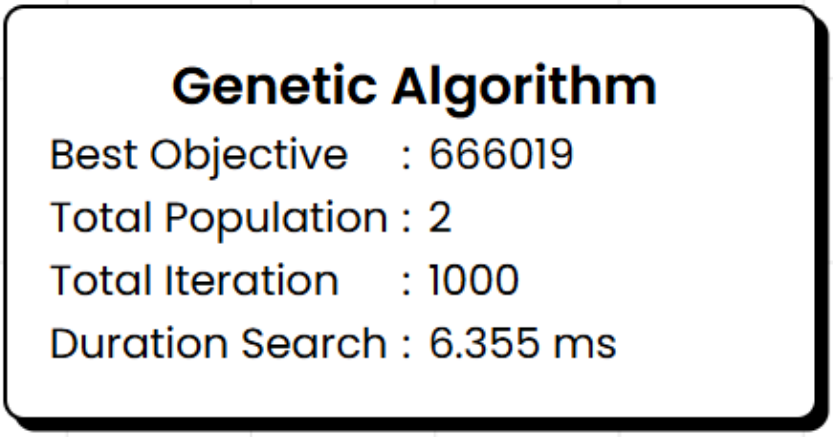
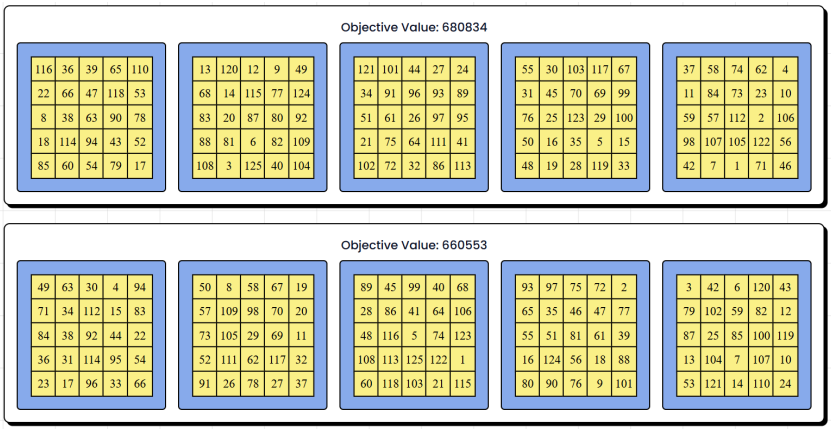
State Akhir	
Best State	
Plot Objective Value terhadap Iterasi	

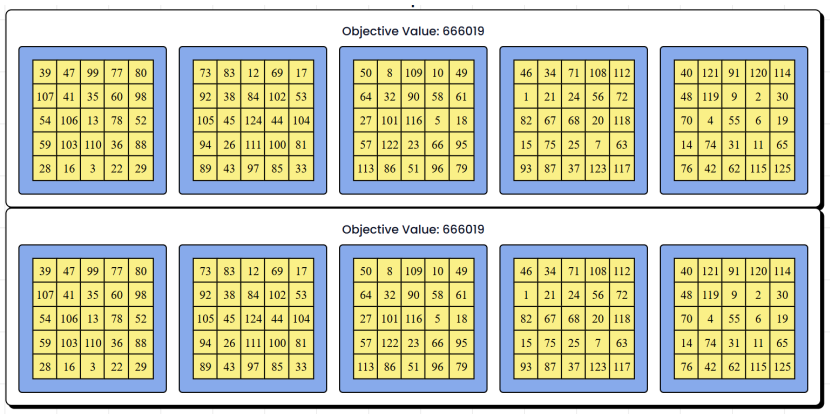
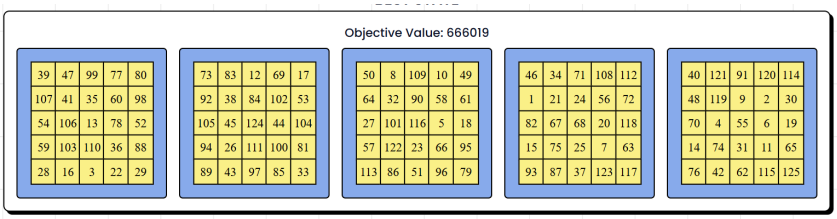
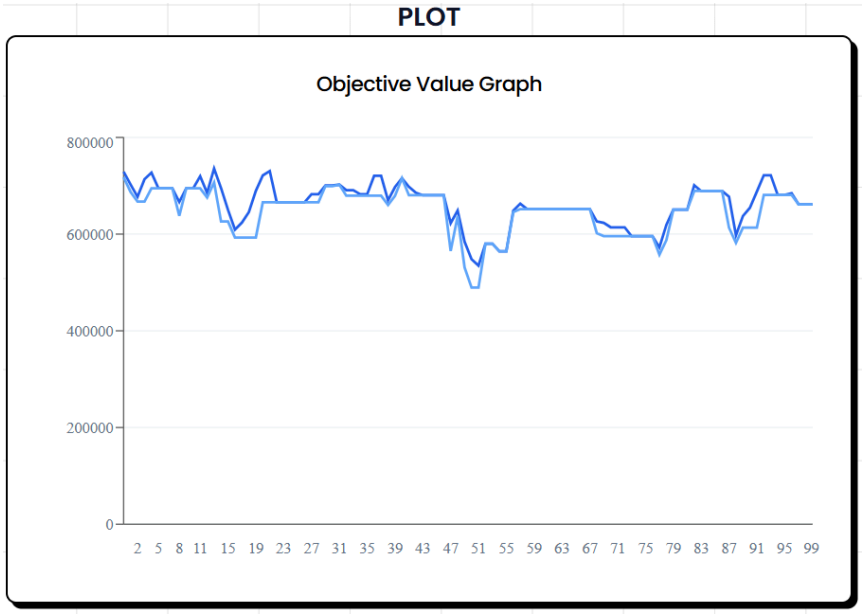
Percobaan 2 : population : 2 & iteration: 100

Objective Value dan Durasi	**Genetic Algorithm** Best Objective : 661511 Total Population : 2 Total Iteration : 100 Duration Search : 0.8 ms
State Awal	Objective Value: 672800 35 15 64 108 9 | 121 95 100 82 1 | 77 106 3 32 86 | 31 37 102 43 114 | 111 57 117 67 69 13 54 29 20 23 | 12 61 83 45 84 | 81 93 90 56 65 | 110 25 116 51 72 | 40 4 118 89 39 124 101 30 63 62 | 103 34 97 98 50 | 99 91 115 66 80 | 10 24 73 8 71 | 48 120 11 33 6 76 16 88 94 53 | 7 58 68 122 18 | 42 119 19 112 21 | 104 105 28 113 5 | 78 49 74 96 92 79 41 52 38 59 | 87 109 85 60 36 | 46 75 125 47 107 | 17 14 27 44 22 | 2 123 70 26 55 Objective Value: 786116 120 51 70 7 23 | 96 93 22 40 92 | 77 16 104 56 9 | 35 102 110 41 52 | 103 86 3 26 5 67 109 62 95 121 | 33 31 106 107 17 | 49 36 108 99 60 | 69 25 34 75 18 | 76 84 97 24 112 124 101 68 115 32 | 57 119 61 45 81 | 66 105 90 72 59 | 88 122 43 85 44 | 28 78 8 21 48 80 117 6 53 113 | 116 71 50 27 47 | 58 74 20 39 82 | 63 54 73 15 89 | 42 38 10 79 1 125 91 87 111 14 | 4 19 98 123 12 | 37 13 2 114 64 | 11 83 29 65 46 | 55 100 94 118 30
State Akhir	Objective Value: 661511 59 101 117 52 36 | 22 41 61 38 79 | 18 11 83 29 103 | 49 13 2 116 93 | 30 95 45 19 24 77 82 107 98 50 | 1 31 37 34 97 | 26 90 106 25 80 | 14 27 65 109 85 | 73 28 69 76 84 99 91 4 108 62 | 40 92 33 123 120 | 42 6 53 111 63 | 60 46 75 125 47 | 78 48 51 72 10 8 71 104 105 57 | 23 67 15 89 66 | 54 112 21 3 32 | 12 16 70 87 7 | 113 56 119 55 100 115 114 110 17 44 | 64 35 124 68 122 | 86 81 9 5 88 | 58 74 20 39 118 | 94 121 96 102 43 Objective Value: 661511 59 101 117 52 36 | 22 41 61 38 79 | 18 11 83 29 103 | 49 13 2 116 93 | 30 95 45 19 24 77 82 107 98 50 | 1 31 37 34 97 | 26 90 106 25 80 | 14 27 65 109 85 | 73 28 69 76 84 99 91 4 108 62 | 40 92 33 123 120 | 42 6 53 111 63 | 60 46 75 125 47 | 78 48 51 72 10 8 71 104 105 57 | 23 67 15 89 66 | 54 112 21 3 32 | 12 16 70 87 7 | 113 56 119 55 100 115 114 110 17 44 | 64 35 124 68 122 | 86 81 9 5 88 | 58 74 20 39 118 | 94 121 96 102 43
Best State	BEST STATE Objective Value: 661511 59 101 117 52 36 | 22 41 61 38 79 | 18 11 83 29 103 | 49 13 2 116 93 | 30 95 45 19 24 77 82 107 98 50 | 1 31 37 34 97 | 26 90 106 25 80 | 14 27 65 109 85 | 73 28 69 76 84 99 91 4 108 62 | 40 92 33 123 120 | 42 6 53 111 63 | 60 46 75 125 47 | 78 48 51 72 10 8 71 104 105 57 | 23 67 15 89 66 | 54 112 21 3 32 | 12 16 70 87 7 | 113 56 119 55 100 115 114 110 17 44 | 64 35 124 68 122 | 86 81 9 5 88 | 58 74 20 39 118 | 94 121 96 102 43

Plot Objective Value terhadap Iterasi	

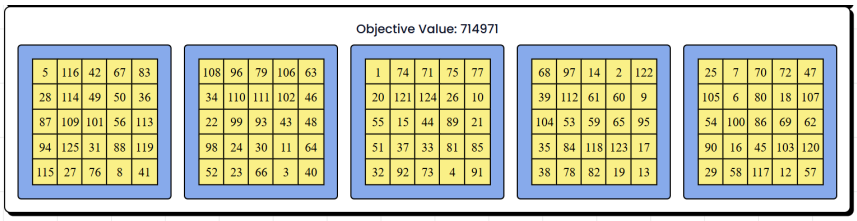
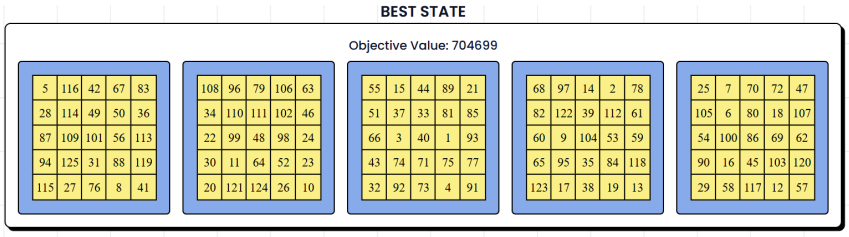
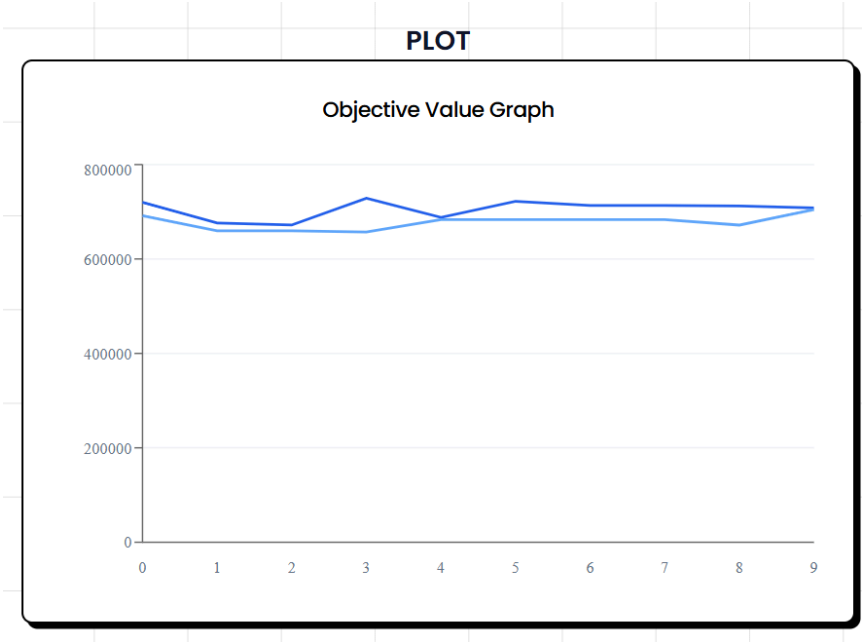
Percobaan 3 : population : 2 & iteration: 1000

Objective Value dan Durasi	
State Awal	

State Akhir	
Best State	
Plot Objective Value terhadap Iterasi	

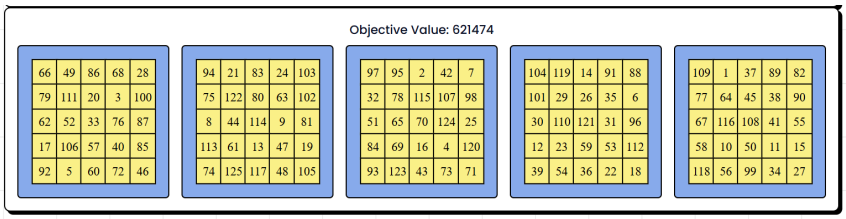
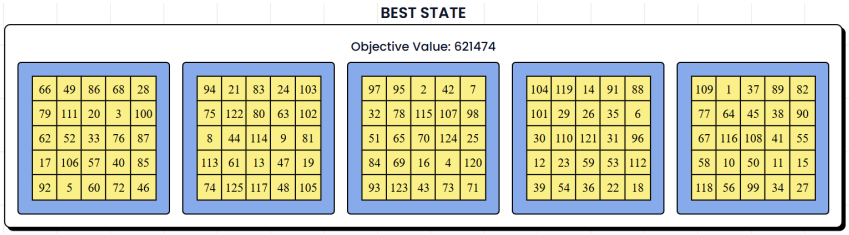
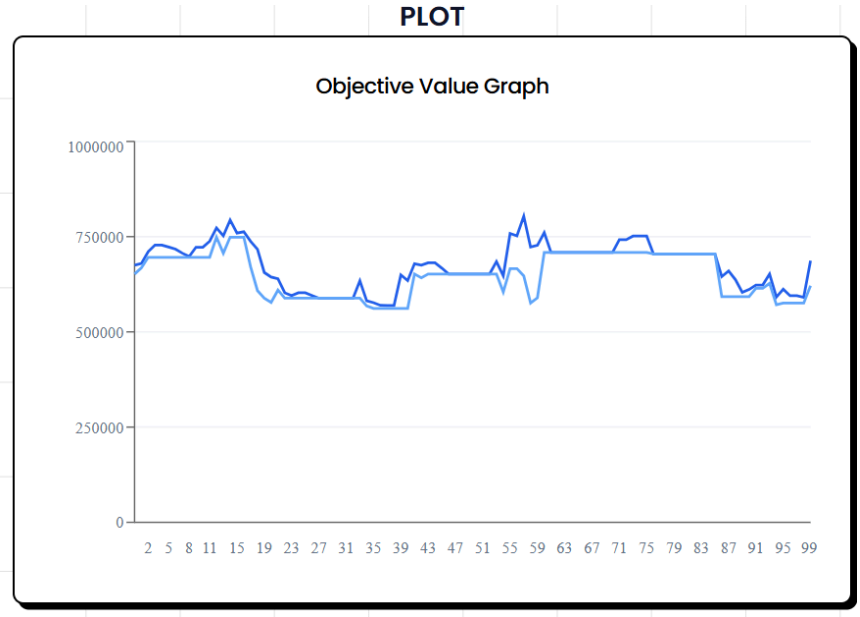
Percobaan 4: population 3 & iteration 10

Objective Value dan Durasi	**Genetic Algorithm** Best Objective : 704699 Total Population : 3 Total Iteration : 10 Duration Search : 0 ms
State Awal	Objective Value: 792237 48 47 8 32 86 85 26 90 17 83 94 45 19 106 38 56 53 87 120 51 13 89 25 15 124 108 59 71 114 103 29 123 60 118 12 125 113 91 102 64 95 46 72 79 75 14 33 116 41 101 122 36 37 23 70 39 78 61 76 57 44 82 111 74 62 105 30 34 52 7 112 27 107 24 117 104 96 16 6 92 81 115 55 5 73 4 88 35 1 69 93 84 43 54 22 100 28 68 77 9 63 99 31 2 18 65 109 98 58 21 11 119 20 49 10 40 50 3 66 80 67 97 42 121 110 Objective Value: 701895 114 107 2 123 95 91 66 78 84 22 93 68 52 105 104 74 65 1 19 51 47 113 8 82 115 99 97 70 96 21 36 34 71 76 54 109 100 33 110 73 40 37 98 119 32 101 87 49 46 89 9 13 121 43 31 118 53 59 86 62 61 90 38 4 16 25 45 35 92 103 120 29 6 58 117 12 57 5 116 42 67 83 28 102 11 77 81 79 94 106 18 14 7 112 48 125 124 30 72 56 88 26 122 41 63 75 3 23 64 55 44 10 69 80 108 85 27 39 17 60 20 24 111 15 50 Objective Value: 772283 116 67 96 79 106 63 62 34 110 111 102 12 46 22 99 48 24 30 11 64 52 5 103 23 20 121 124 83 26 10 55 15 44 89 21 51 37 120 33 58 81 85 29 66 57 117 3 40 1 16 93 43 74 71 90 65 95 35 84 118 123 17 6 38 78 82 19 13 25 7 70 72 47 105 80 18 107 54 100 86 69 98 114 49 75 77 32 92 73 4 91 68 97 14 2 122 39 112 50 53 36 28 87 109 101 56 45 113 94 125 59 61 60 9 104 31 88 119 115 27 76 42 8 41 108
State Akhir	Final Population Objective Value: 704699 5 116 42 67 83 28 114 49 50 36 87 109 101 56 113 94 125 31 88 119 115 27 76 8 41 108 96 79 106 63 34 110 111 102 46 22 99 48 98 24 30 11 64 52 23 20 121 124 26 10 55 15 44 89 21 51 37 33 81 85 66 3 40 1 93 43 74 71 75 77 32 92 73 4 91 68 97 14 2 78 82 122 39 112 61 60 9 104 53 59 65 95 35 84 118 123 17 38 19 13 25 7 70 72 47 105 6 80 18 107 54 100 86 69 62 90 16 45 103 120 29 58 117 12 57 Objective Value: 704699 5 116 42 67 83 28 114 49 50 36 87 109 101 56 113 94 125 31 88 119 115 27 76 8 41 108 96 79 106 63 34 110 111 102 46 22 99 48 98 24 30 11 64 52 23 20 121 124 26 10 55 15 44 89 21 51 37 33 81 85 66 3 40 1 93 43 74 71 75 77 32 92 73 4 91 68 97 14 2 78 82 122 39 112 61 60 9 104 53 59 65 95 35 84 118 123 17 38 19 13 25 7 70 72 47 105 6 80 18 107 54 100 86 69 62 90 16 45 103 120 29 58 117 12 57

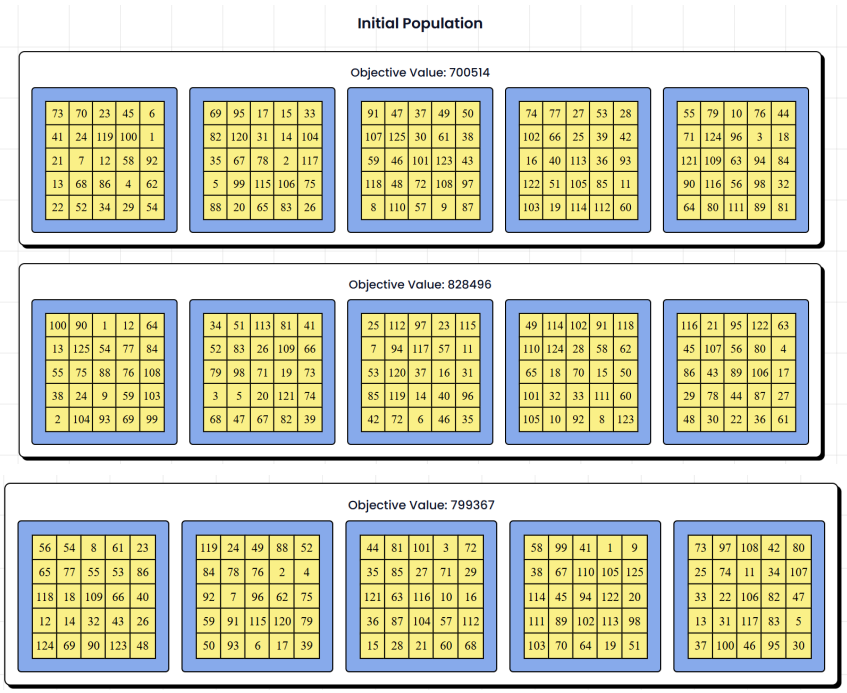
	
Best State	
Plot Objective Value terhadap Iterasi	

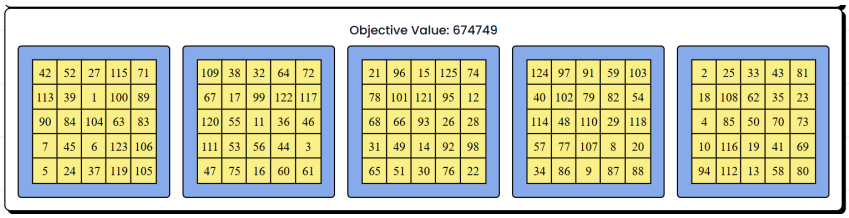
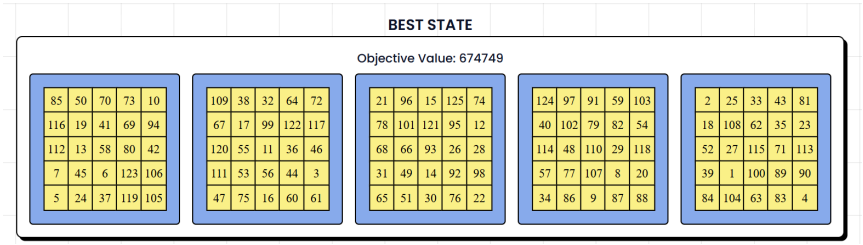
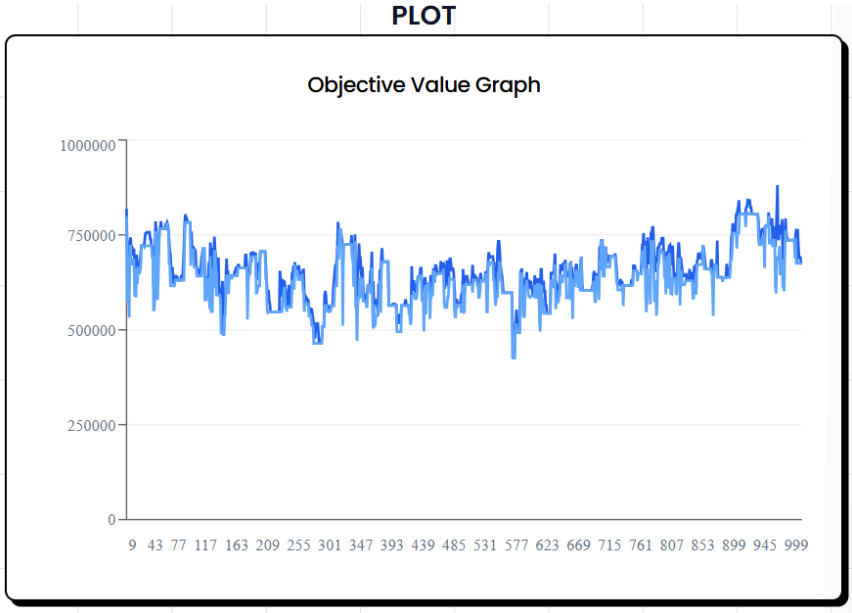
Percobaan 5: population 3 & iteration 100

Objective Value dan Durasi	**Genetic Algorithm** Best Objective : 621474 Total Population : 3 Total Iteration : 100 Duration Search : 0.581 ms
State Awal	**Initial Population** Objective Value: 651742 111 100 3 52 79 / 62 28 8 102 81 / 113 89 82 77 68 / 33 76 64 45 66 / 26 25 67 1 53 97 12 48 65 87 / 86 23 124 59 107 / 6 84 49 123 69 / 30 78 29 70 98 / 37 35 96 51 105 115 112 43 73 104 / 71 38 40 114 9 / 85 92 5 60 17 / 83 90 116 108 24 / 103 75 72 39 54 36 15 41 110 121 / 31 122 80 63 22 / 18 109 95 2 42 / 7 32 119 14 91 / 88 101 61 13 47 19 74 125 117 16 / 4 120 93 46 94 / 21 118 56 99 34 / 27 106 57 55 58 / 10 50 11 20 44 Objective Value: 648734 118 73 69 92 93 / 111 40 26 15 103 / 113 119 12 87 14 / 32 31 16 78 27 / 106 42 122 23 54 76 39 75 90 20 / 53 110 25 5 51 / 125 3 2 114 36 / 22 9 109 59 46 / 29 97 82 30 56 80 6 100 11 117 / 34 71 45 99 4 / 38 88 81 62 115 / 58 85 60 64 52 / 66 89 47 96 124 70 94 108 77 24 / 102 107 35 98 55 / 33 19 57 8 7 / 72 44 104 43 95 / 49 120 101 28 41 74 91 105 112 123 / 121 10 84 68 65 / 67 116 63 37 83 / 18 79 48 21 86 / 61 13 17 50 1 Objective Value: 674011 52 49 62 11 47 / 15 50 86 110 109 / 45 95 60 73 38 / 87 1 125 72 37 / 67 108 94 63 58 103 89 5 118 88 / 78 20 76 115 44 / 8 85 71 123 93 / 9 22 48 119 105 / 27 97 29 43 80 41 3 26 92 99 / 14 40 35 21 106 / 46 6 2 64 54 / 30 57 96 12 66 / 23 82 16 91 117 101 90 74 68 39 / 59 53 112 19 107 / 98 51 121 65 122 / 34 79 111 13 81 / 114 61 4 102 28 120 10 70 100 56 / 55 124 83 25 31 / 32 17 113 75 42 / 84 33 36 24 77 / 104 7 116 18 69
State Akhir	**Final Population** Objective Value: 707818 66 49 86 68 28 / 79 111 20 3 100 / 62 52 33 76 87 / 17 106 57 40 85 / 92 5 60 72 46 94 21 83 24 103 / 75 122 80 63 61 / 13 47 19 74 125 / 117 48 105 97 95 / 2 42 7 32 78 115 107 98 51 65 / 70 124 25 84 69 / 16 4 120 93 123 / 43 73 71 104 119 / 14 91 88 101 29 102 8 44 114 9 / 81 113 26 35 6 / 30 110 121 31 96 / 12 23 59 53 112 / 39 54 36 22 18 109 1 37 89 82 / 77 64 45 38 90 / 67 116 108 41 55 / 58 10 50 11 15 / 118 56 99 34 27 Objective Value: 731643 79 111 20 3 100 / 62 52 33 76 87 / 17 106 57 40 85 / 92 5 60 72 46 / 94 21 83 24 103 31 96 12 23 59 / 75 122 80 63 61 / 13 47 19 74 125 / 117 48 105 97 95 / 2 42 7 32 78 115 107 98 51 65 / 70 124 25 84 69 / 16 4 120 93 123 / 43 73 71 104 119 / 14 91 88 101 29 26 35 6 30 110 / 121 53 112 39 54 / 36 22 18 109 1 / 37 102 8 44 114 / 9 81 113 89 82 77 64 45 38 90 / 67 116 108 41 55 / 58 10 50 11 15 / 118 56 99 34 27 / 66 49 86 68 28

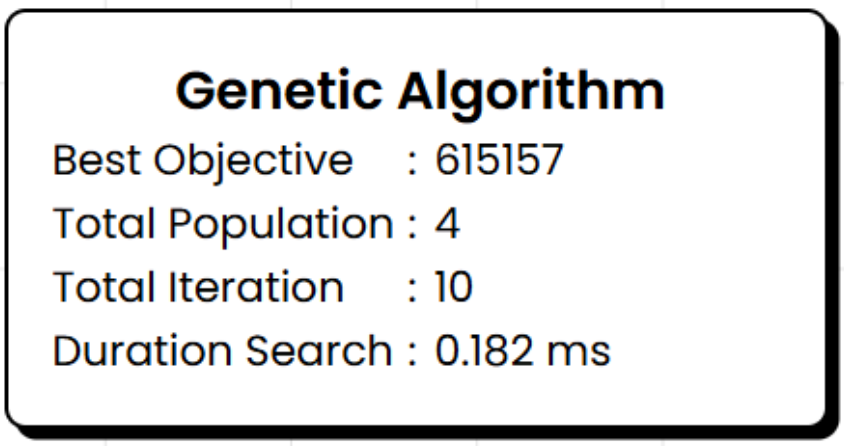
	
Best State	
Plot Objective Value terhadap Iterasi	

Percobaan 6: population 3 & iteration 1000

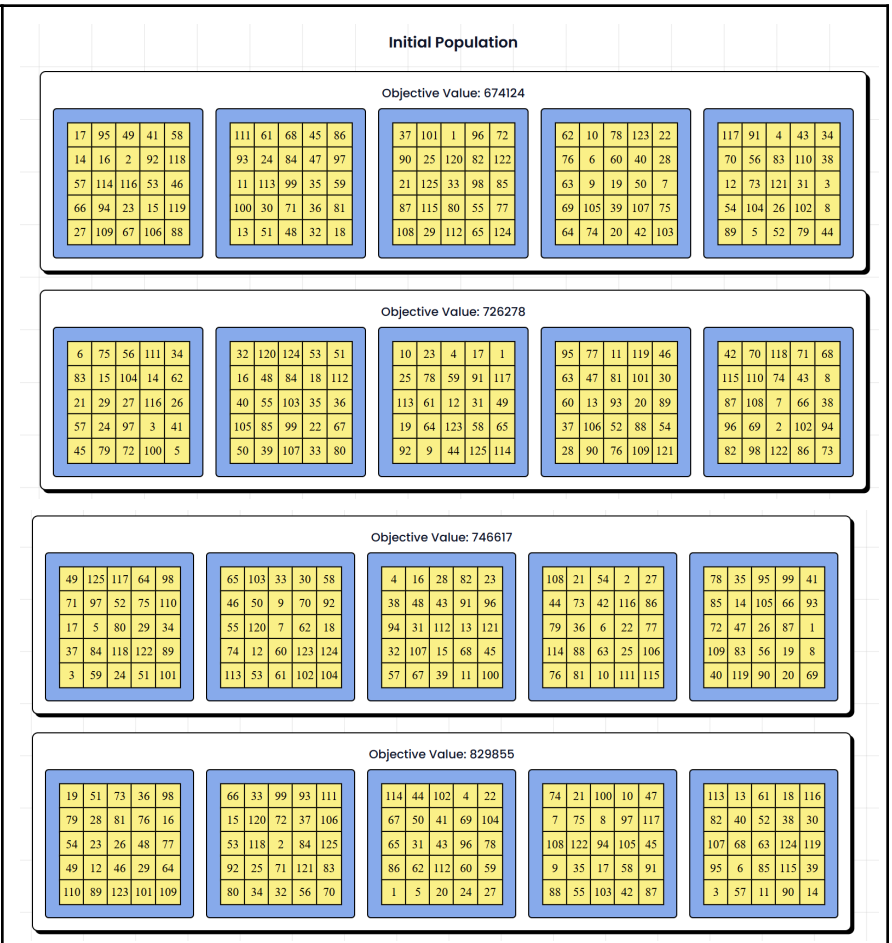
Objective Value dan Durasi	**Genetic Algorithm** Best Objective : 674749 Total Population : 3 Total Iteration : 1000 Duration Search : 7.601 ms
State Awal	
State Akhir	

	
Best State	
Plot Objective Value terhadap Iterasi	

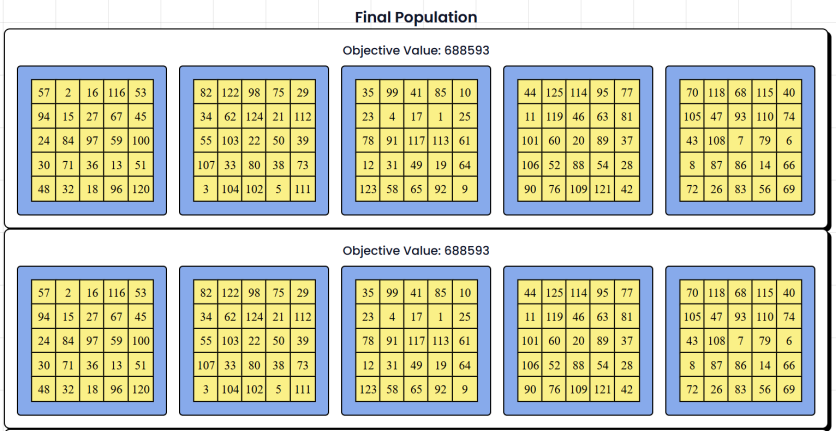
Percobaan 7: population 4 & iteration 10

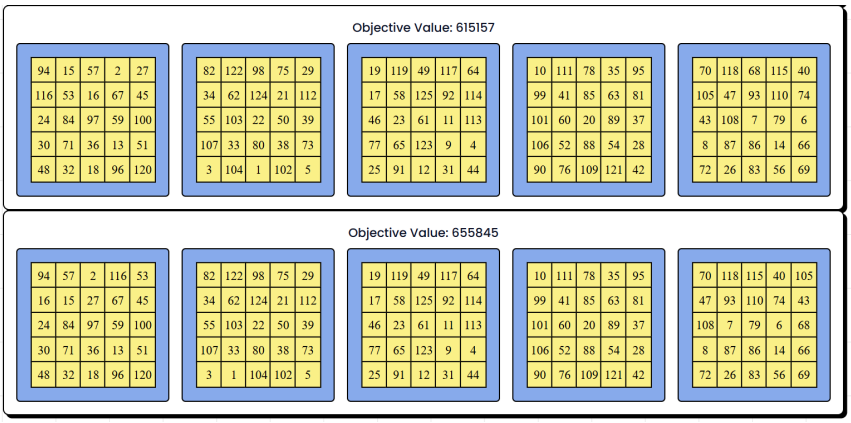
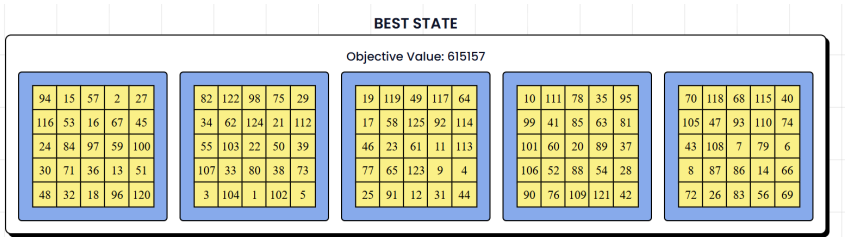
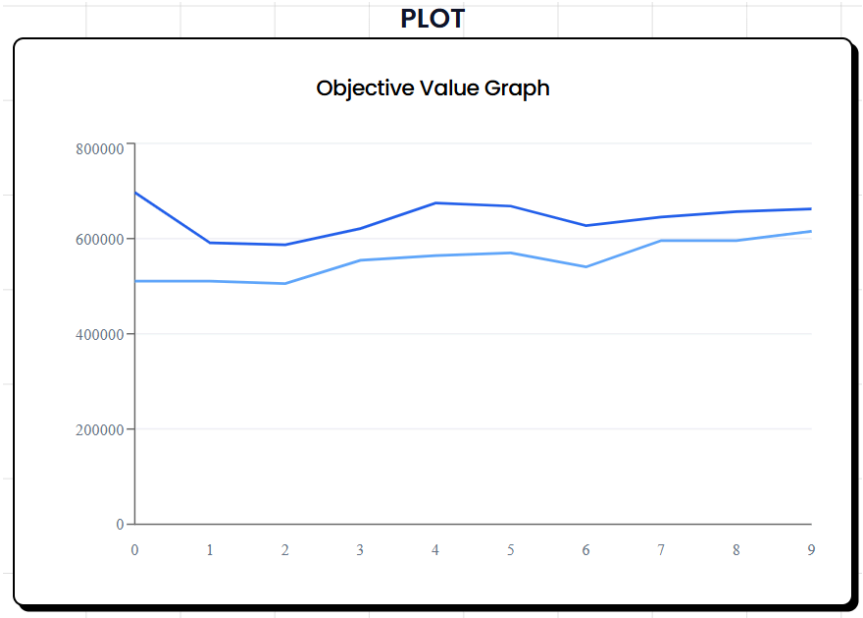
Objective Value dan Durasi	

State Awal



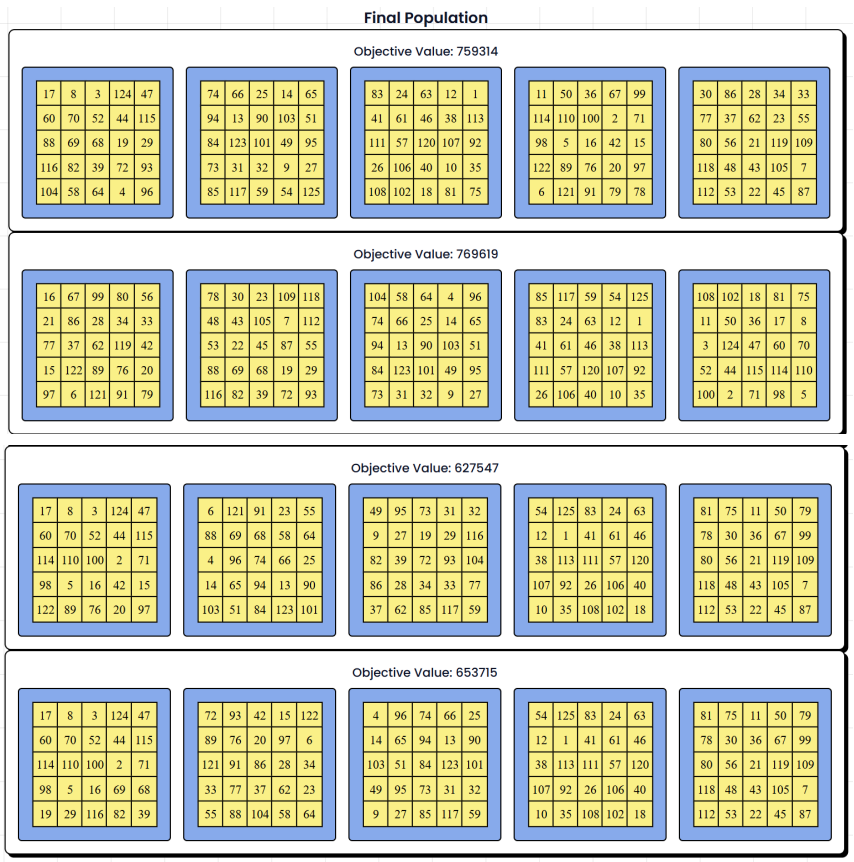
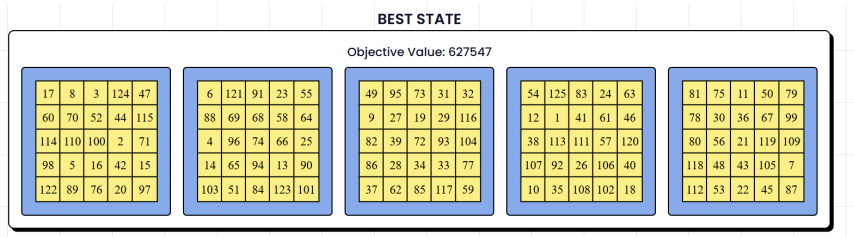
State Akhir

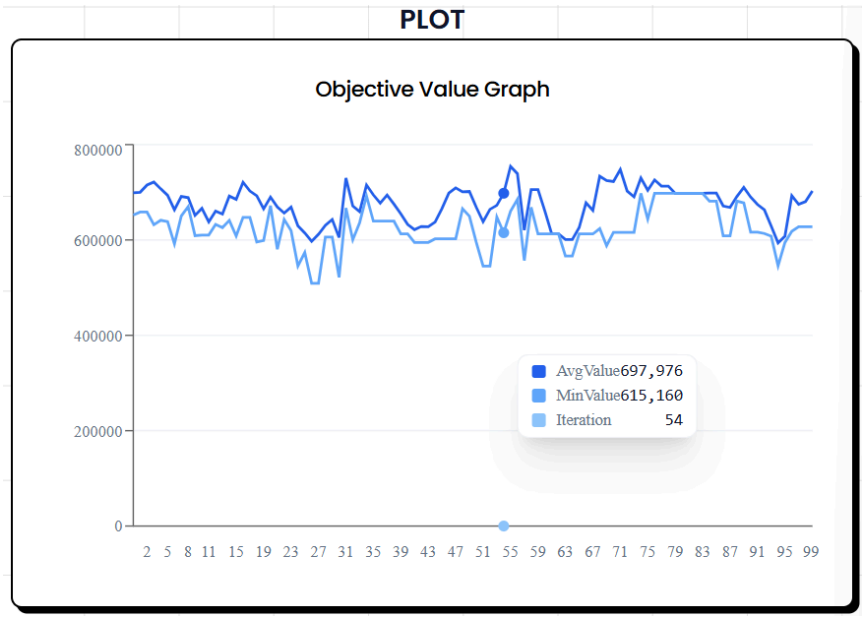


	
Best State	
Plot Objective Value terhadap Iterasi	

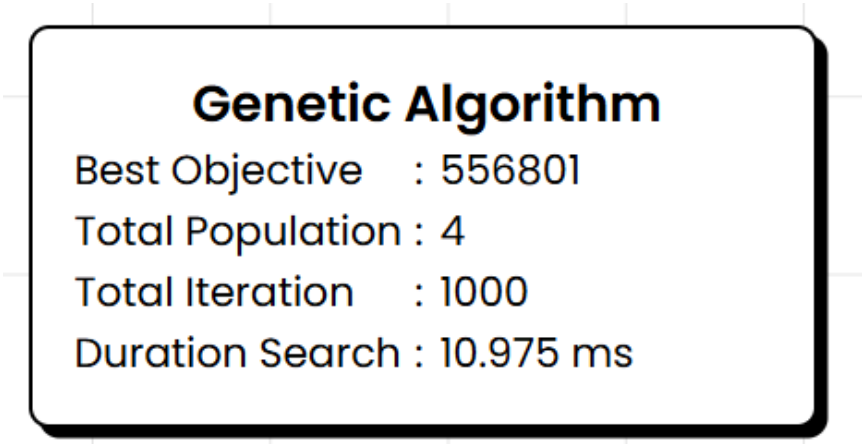
Percobaan 8: population 4 & iteration 100

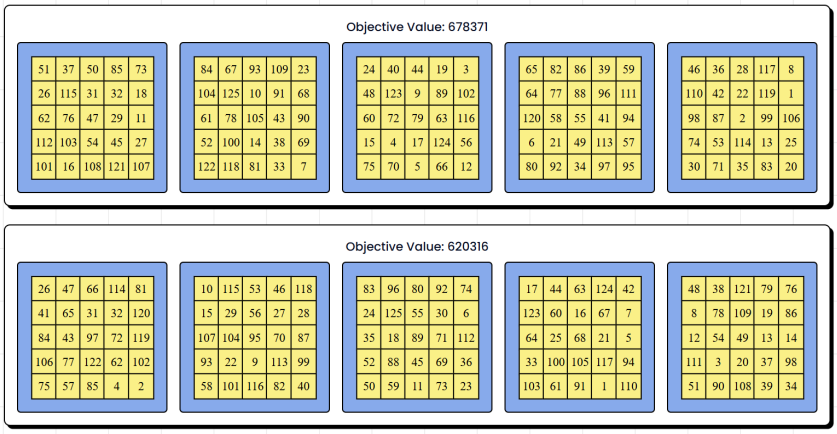
Objective Value dan Durasi	**Genetic Algorithm** Best Objective : 627547 Total Population : 4 Total Iteration : 100 Duration Search : 1.021 ms
State Awal	**Initial Population** Objective Value: 731764 52 44 115 120 2 / 71 98 5 88 3 / 82 77 51 123 101 / 70 87 104 61 46 / 58 9 16 83 84 64 110 107 62 117 / 49 20 11 17 100 / 74 103 66 109 93 / 23 76 29 89 86 / 96 68 28 8 34 33 27 37 59 113 / 108 55 24 54 38 / 56 69 122 119 90 / 39 32 118 48 43 / 12 18 50 81 31 111 125 57 1 41 / 47 60 85 99 114 / 75 92 21 40 80 / 26 4 65 94 116 / 42 15 13 72 105 22 10 35 25 14 / 124 102 106 19 63 / 95 73 97 6 121 / 91 79 78 30 36 / 67 7 112 53 45 Objective Value: 656866 49 65 2 45 77 / 105 61 109 76 21 / 25 64 70 55 54 / 9 87 111 125 74 / 104 110 51 5 113 124 29 75 100 41 / 86 115 16 94 3 / 47 27 107 43 71 / 89 50 1 68 91 / 13 82 7 32 40 11 114 28 42 119 / 20 30 92 17 24 / 102 117 103 95 15 / 73 44 22 26 120 / 48 83 106 39 10 14 58 108 101 96 / 4 116 36 121 79 / 18 81 31 99 72 / 123 69 34 80 85 / 19 97 33 62 12 53 67 84 122 59 / 52 60 23 66 112 / 63 6 57 93 8 / 88 46 78 35 38 / 56 98 118 90 37 Objective Value: 690962 125 34 107 74 53 / 108 103 83 23 29 / 48 50 62 115 45 / 54 112 13 18 77 / 82 78 6 19 121 114 63 79 120 40 / 69 111 4 8 33 / 101 58 95 41 109 / 49 97 57 110 21 / 68 99 46 106 39 16 84 24 7 5 / 124 70 116 30 9 / 92 43 56 36 66 / 113 28 117 42 76 / 25 90 122 105 89 64 17 72 32 3 / 87 104 38 44 86 / 71 22 37 75 52 / 96 67 35 20 59 / 55 123 98 15 47 31 85 118 119 11 / 100 12 73 94 65 / 27 93 102 14 81 / 10 60 80 26 2 / 1 91 51 88 61 Objective Value: 714470 25 75 105 7 8 / 124 62 125 98 19 / 66 4 35 71 118 / 76 60 103 11 65 / 116 90 49 79 94 38 3 68 73 123 / 40 1 30 58 55 / 104 113 101 93 26 / 112 42 15 88 77 / 122 100 16 41 107 48 6 52 74 43 / 2 36 61 56 89 / 45 32 9 117 39 / 10 83 24 99 23 / 108 63 114 110 5 64 106 51 80 18 / 119 50 81 84 87 / 47 20 27 95 102 / 29 46 33 44 67 / 86 21 22 121 91 78 54 12 53 85 / 82 37 72 14 69 / 59 120 96 115 57 / 109 34 70 28 111 / 92 17 31 97 13

State Akhir	
Best State	

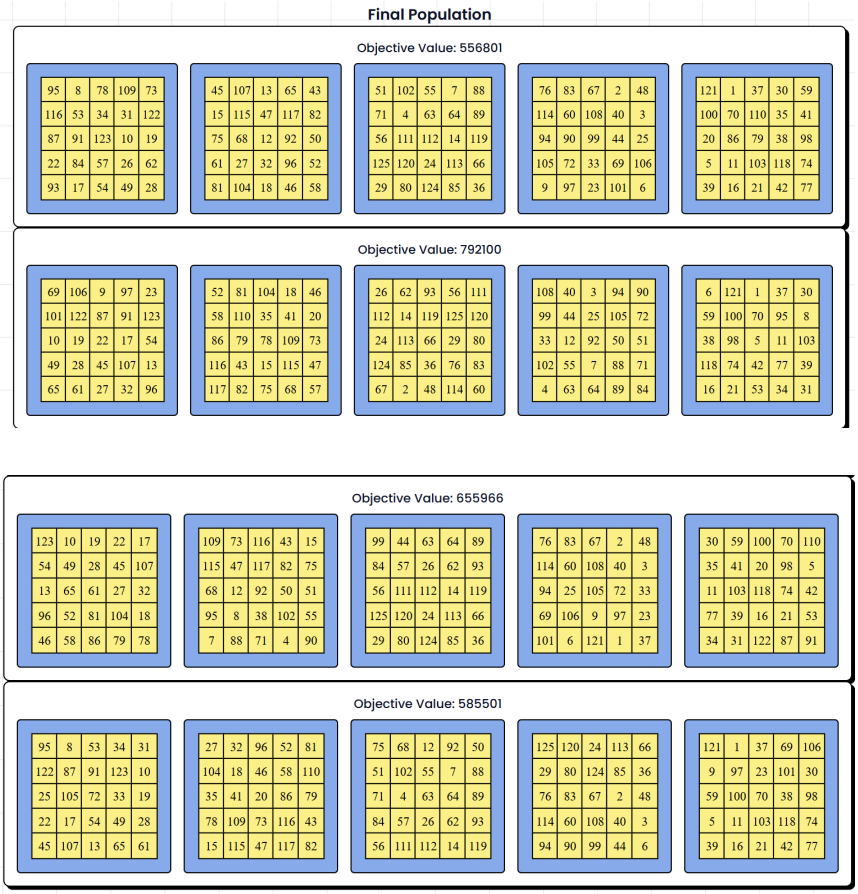
Plot Objective Value terhadap Iterasi	

Percobaan 9: population 4 & iteration 1000

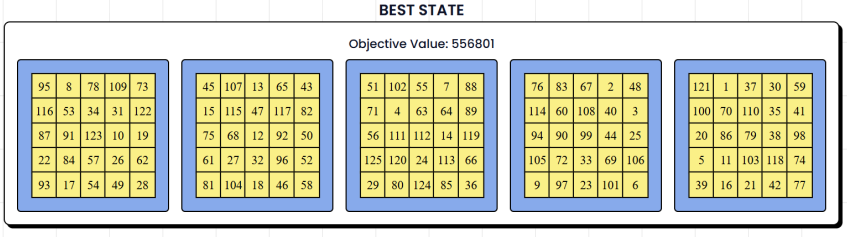
Objective Value dan Durasi	
State Awal	

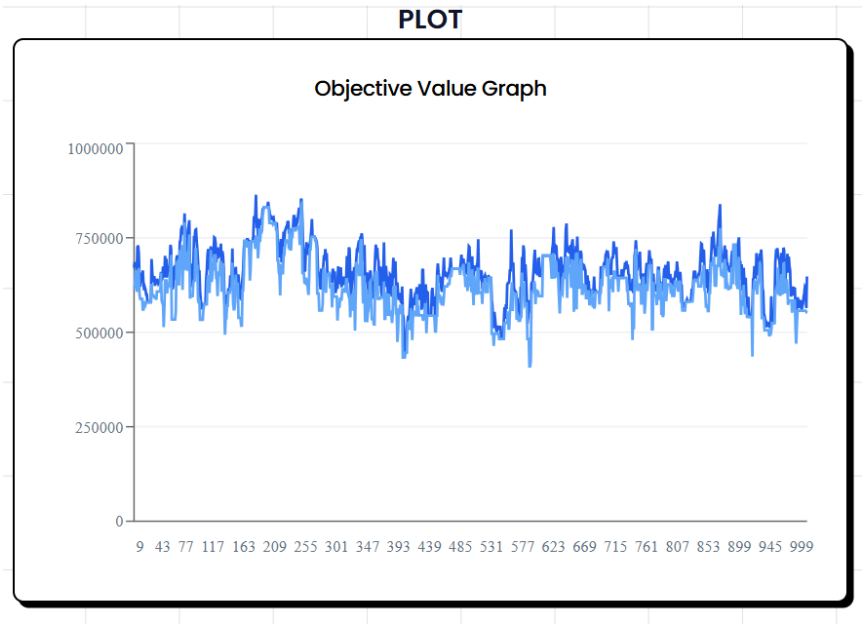


State Akhir



Best State



Plot Objective Value terhadap Iterasi	

BAB V
ANALISIS DAN PEMBAHASAN

Berdasarkan eksperimen yang dilakukan terhadap setiap algoritma pencarian lokal pada bagian sebelumnya, berikut ini disajikan tabel nilai *objective value* dari state terbaik yang diperoleh oleh setiap algoritma:

Algoritma	**Objective Value**	**Average Time (ms)**
Steepest Ascent Hill Climbing	188	1198,485
Hill-climbing with Sideway	117	1257,841
Hill-climbing with Random Restart	128	3380 – 4990 (tergantung banyak restart)
Stochastic Hill-climbing	205	1122,331
Simulated Annealing	58	632,449
Genetic Algorithm	556801	0 – 6000 (tergantung banyak

		iterasi)

Berdasarkan data pada tabel tersebut, terlihat bahwa algoritma *Simulated Annealing* memiliki *objective value* paling mendekati *global objective value*, yaitu 0. Algoritma ini juga selalu konsisten memberikan hasil terbaik di antara algoritma lainnya. Hal ini terjadi karena mekanisme probabilistiknya yang memungkinkan perpindahan ke *neighbor* dengan *objective value* yang lebih buruk berdasarkan temperatur yang perlahan-lahan menurun. Dengan karakteristik ini, *Simulated Annealing* dapat keluar dari jebakan *local maxima*, sehingga memperbesar kemungkinan mencapai solusi yang lebih optimal. Selain itu, algoritma ini hanya perlu menghasilkan *neighbor* secara acak sehingga waktu eksekusinya relatif cepat. Meski demikian, algoritma ini memerlukan waktu *convergence* yang cukup lama karena harus menjalankan sejumlah besar iterasi, khususnya pada temperatur rendah. Peningkatan pada *cooling rate* juga dapat meningkatkan kualitas solusi, tetapi hal ini akan meningkatkan biaya komputasi karena jumlah iterasi yang lebih banyak.

Algoritma *Hill-climbing with Sideway* menempati posisi kedua dalam hal *objective value* terbaik. Algoritma ini bekerja dengan berpindah ke *state neighbor* yang memiliki nilai *objective value* lebih kecil atau sama dengan *state* saat ini, sehingga dapat mengatasi kendala *local maxima*. Strategi ini memberikan fleksibilitas lebih dibandingkan *Steepest Ascent* karena memungkinkan pergerakan pada nilai yang sama (*sideway move*), tetapi terdapat risiko algoritma terjebak di *local plateau*—suatu wilayah solusi di mana nilai *objective value* stagnan—sehingga algoritma berpotensi berjalan tanpa batas. Untuk mengatasi hal ini, diperlukan pembatasan jumlah pergerakan *sideway* secara berurutan. Secara umum, algoritma ini memiliki waktu eksekusi yang lebih lama dibandingkan *Simulated Annealing* karena mengevaluasi semua *neighbor*.

Selanjutnya, algoritma *Hill-climbing with Random Restart* menduduki posisi ketiga dalam pencapaian *objective value*. Algoritma ini memperbaiki kelemahan *Steepest Ascent* melalui strategi *random restart*, yakni memulai pencarian dari beberapa titik awal yang berbeda secara acak. Hal ini meningkatkan kemungkinan menemukan solusi yang lebih baik, terutama dalam kasus masalah yang memiliki banyak *local maxima*. Namun, kelemahan algoritma ini adalah ketergantungannya pada *random initial state*, yang membuat waktu *convergence* sangat bervariasi tergantung jumlah *restart* dan karakteristik dari *state* awal yang dipilih.

Algoritma *Steepest Ascent Hill Climbing* menghasilkan *objective value* terbaik keempat dan memiliki waktu *convergence* paling cepat di antara algoritma yang diuji. Algoritma ini selalu memilih *neighbor* dengan *objective value* terbaik pada setiap iterasi, yang membuatnya cenderung lebih efisien dari segi waktu. Namun, algoritma ini mudah terjebak di *local maxima* karena tidak memiliki mekanisme untuk keluar dari jebakan tersebut, sehingga kinerjanya terbatas pada struktur lanskap solusi.

Pada posisi selanjutnya, *Stochastic Hill-climbing* memiliki *objective value* yang relatif buruk karena pergerakannya yang cenderung acak, sehingga kurang konsisten dalam menemukan solusi yang optimal. Algoritma ini hanya berpindah ke *neighbor* dengan *objective value* lebih kecil secara acak, sehingga memiliki kemungkinan tinggi terjebak pada *local maxima*. Namun, dari segi waktu

eksekusi, algoritma ini relatif cepat karena menggunakan mekanisme pembangkitan *neighbor* secara acak.

Genetic Algorithm (GA) memiliki karakteristik unik dibandingkan algoritma pencarian lokal lainnya, terutama karena pendekatan berbasis populasi dan prinsip evolusi yang diusungnya. Dengan menggunakan mekanisme crossover dan mutation, GA mampu mengeksplorasi ruang solusi yang lebih luas dan memperkenalkan variasi di setiap generasi baru. Namun, algoritma ini tidak selalu menghasilkan populasi dengan objective value yang optimal, karena proses crossover dan mutation dilakukan secara acak. Selain itu, performa GA sangat bergantung pada jenis masalah yang ingin dipecahkan. Dalam kasus tertentu, algoritma ini mungkin kurang efektif. Hal ini terlihat jelas pada tabel di atas, dimana GA memiliki best objective value yang paling buruk.

Hal lain yang perlu diperhatikan adalah kesulitan untuk mencapai *global objective value* karena ruang pencarian dalam masalah ini sangat besar, yaitu 125!, sementara hanya ada satu konfigurasi solusi yang benar (dengan beberapa susunan lainnya yang simetris atau identik dalam perspektif kubus yang berbeda). Ruang pencarian yang besar ini juga menyebabkan adanya banyak *local maxima* yang dapat menjebak algoritma pencarian lokal, sehingga algoritma rentan berhenti pada solusi yang tidak optimal.

KESIMPULAN

Berdasarkan eksperimen dan analisis yang telah dilakukan, algoritma pencarian lokal yang menunjukkan kinerja terbaik dalam hal *objective value* dan waktu eksekusi adalah ***Simulated Annealing***. Algoritma ini secara konsisten mampu menemukan solusi mendekati optimal dalam waktu yang relatif cepat, menjadikannya pilihan yang efisien untuk berbagai jenis masalah pencarian lokal. Di sisi lain, algoritma dengan performa terburuk dari segi *objective value* dalam percobaan ini adalah *Genetic Algorithm*. Hal ini disebabkan oleh sifat *crossover* dan *mutation* yang acak, yang membuat GA cenderung mengalami fluktuasi performa tergantung pada masalah yang dipecahkan dan parameter yang digunakan.

r

REFERENSI

E. W. Weisstein, "Magic Cube," *MathWorld*. [Online]. Available: https://mathworld.wolfram.com/MagicCube.html. [Accessed: Oct. 2, 2024].

E. W. Weisstein, "Perfect Magic Cube," *MathWorld*. [Online]. Available: https://mathworld.wolfram.com/PerfectMagicCube.html. [Accessed: Oct. 2, 2024].

Planète Maths, "Perfect Magic Cubes." [Online]. Available: http://www.multimagie.com/English/Perfectcubes.htm. [Accessed: Oct. 2, 2024].

S. Russell and P. Norvig, *Artificial Intelligence: A Modern Approach*, 4th ed. Pearson Higher Education, 2019.

Dr. Masayu Leylia Khodra, S.T, M.T., "Artificial Intelligence," Class lecture, 2024. [PowerPoint slides].

PEMBAGIAN TUGAS

NIM	Task
13522068	Server (State, Genetic Algorithm, API setup, Handler for Genetic Algorithm), Report
13522093	Clients (UI) + Video Player, Report
13522098	Server (Hill Climbing, Simulated Annealing, Handler for Hill Climbing ada Simulated Annealing)
13522118	Clients (UI) + Video Player, Report