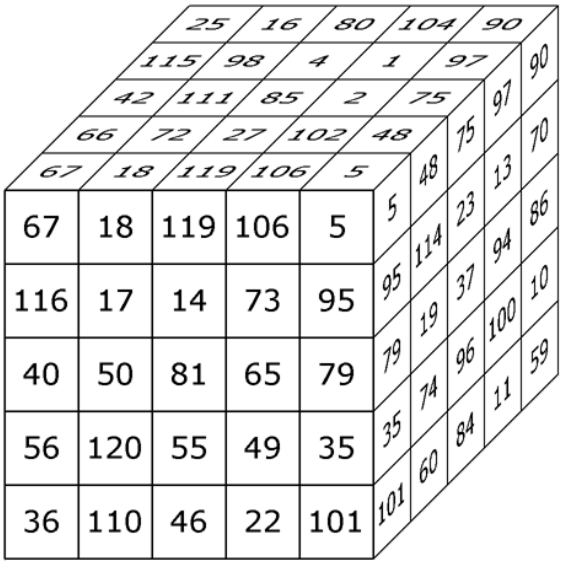
Tugas Besar 1 IF3070 Dasar Inteligensi Artifisial
Pencarian Solusi Diagonal Magic Cube dengan Local Search



Dipersiapkan oleh

Kelompok 55

18222034	Christoper Daniel
18222035	Lydia Gracia
18222049	Willhelmina Rachel Silalahi
18222100	Ervina Limka

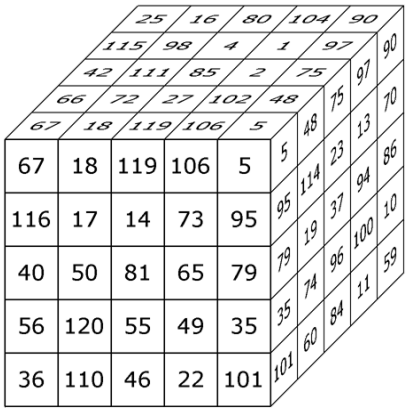
Awal pengerjaan: 8 Oktober 2024

Akhir pengerjaan: 11 November 2024 23:59 WIB

DAFTAR ISI

DESKRIPSI PERSOALAN **3**
PEMBAHASAN **5**
I. Pemilihan Objective Function 5
II. Penjelasan Implementasi Algoritma Local Search 7
III. Hasil Eksperimen dan Analisis 39
KESIMPULAN DAN SARAN **61**
I. Kesimpulan 61
II. Saran 61
PEMBAGIAN TUGAS **62**
REFERENSI **63**

DESKRIPSI PERSOALAN



Diagonal magic cube merupakan sebuah permasalahan yang melibatkan pencarian solusi untuk menghasilkan sebuah permutasi dari n^3 buah angka yang memenuhi syarat tertentu. Dalam konteks ini, n adalah sisi dari kubus yang dianggap sebagai n x n x n. Tujuan dari permasalahan ini adalah mengatur setiap elemen dalam kubus sehingga setiap diagonal memiliki nilai yang sama. Angka-angka yang tersusun pada kubus harus memenuhi persyaratan sebagai berikut.

- Terdapat satu angka magic number dari kubus tersebut
- Jumlah angka-angka untuk setiap baris sama dengan magic number
- Jumlah angka-angka untuk setiap kolom sama dengan magic number
- Jumlah angka-angka untuk setiap tiang sama dengan magic number
- Jumlah angka-angka untuk seluruh diagonal ruang dan potongan bidang pada kubus sama dengan magic number

Local Search adalah suatu metode dalam algoritma pencarian yang memulai dari suatu solusi awal dan mencoba untuk memperbaiki melalui perubahan kecil yang diharapkan menghasilkan solusi yang lebih baik. Ada 6 jenis local search yang digunakan untuk mencari solusi, antara lain:

1. Steepest Ascent Hill-climbing
2. Hill-climbing with Sideways Move
3. Random Restart Hill-climbing
4. Stochastic Hill-climbing
5. Simulated Annealing

6. Genetic Algorithm

Metode ini biasanya digunakan dalam optimisasi dan mencari solusi yang lebih optimal dalam berbagai permasalahan. Terdapat beberapa langkah pada pencarian solusi diagonal *magic cube* dengan *local search* yaitu sebagai berikut.

1. **Inisialisasi**: memilih atau menghasilkan solusi awal secara acak atau sesuai aturan tertentu.
2. **Evaluasi**: menghitung jumlah angka-angka setiap baris, kolom, tiang, diagonal ruang, dan seluruh diagonal dari solusi yang dihasilkan.
3. **Pencarian lokal**: melakukan perubahan kecil berdasarkan solusi yang ada dengan cara mengganti posisi beberapa elemen untuk menghasilkan nilai yang sama atau lebih dekat dari nilai yang diinginkan.
4. **Perbandingan**: melakukan pengecekan mengenai perubahan apakah meningkatkan atau memperburuk solusi.
5. **Iterasi**: melakukan iterasi pada langkah-langkah di atas sampai tercapai kriteria berhenti, seperti jumlah iterasi tertentu atau solusi sudah optimal.
6. **Terminasi**: proses akan berhenti ketika tidak ada perubahan yang dapat meningkatkan solusi atau telah mencapai iterasi maksimum.

Pencarian lokal memiliki kelebihan seperti mudah untuk diimplementasikan dan cenderung memberikan solusi yang cukup baik dalam waktu singkat. Namun, algoritma hill-climbing ini memiliki kelemahan seperti kemungkinan terjebak pada lokal optimum (solusi yang tidak optimal secara global tetapi sudah optimal di sekitar area yang dicari). Untuk mengatasi hal tersebut, diperlukan menggunakan algoritma lain seperti genetic algorithm ataupun simulated annealing untuk menemukan solusi yang lebih optimal secara global.

Pada tugas ini, akan dilakukan penyelesaian permasalahan Diagonal Magic Cube berukuran 5x5x5 dengan inisial state kubus terdiri dari susunan angka 1 hingga 5^3 secara acak. Setiap iterasi pada algoritma *local search*, langkah yang diperbolehkan adalah menukar posisi 2 angka pada kubus. Khusus untuk *genetic algorithm* dapat dilakukan penukaran posisi lebih dari 2 angka dalam satu iterasi.

PEMBAHASAN

Berikut adalah langkah-langkah yang harus dipenuhi dalam proses pencarian masalah *magic number* ini

- Memilih *objective function* yang akan digunakan untuk menjadi pengukuran seberapa bagus konfigurasi angka yang dihasilkan.
- Mengimplementasikan setiap *local search* untuk menyelesaikan masalah diagonal magic number dengan mempertimbangkan *feasibility* dari setiap solusi.
- Menganalisis dan menentukan algoritma *local search* mana yang paling efektif dan efisien dalam pencapaian solusi.
- Menganalisis setiap kekurangan dan kelebihan pendekatan penyelesaian dengan *local search* maupun *complete search* pada konteks permasalahan ini.
- Membuat rencana implementasi berupa kelas dan fungsi yang mencakup seluruh algoritma yang digunakan.

I. Pemilihan Objective Function

Objective function adalah fungsi untuk menilai suatu keadaan objek yang dibandingkan dengan tujuan pencarian. Pada permasalahan diagonal *magic cube*, *objective function* digunakan untuk menilai seberapa baik susunan angka dalam kubus untuk memenuhi target *magic number*. Oleh karena itu, diperlukan *objective function* untuk mengevaluasi seberapa jauh susunan angka dalam kubus tersebut menyimpang dari *magic number*.

Terdapat tiga pendekatan evaluasi untuk mengevaluasi penyimpangan tersebut:

1. **Menghitung berapa kali *magic number* dipenuhi dalam susunan kubus**

   Cara ini dapat diekspresikan secara matematis dengan pendekatan ***cost based*** dan ***value based***. Pendekatan yang dipilih adalah pendekatan ***cost based*** untuk keseragaman antara rencana *objective function* ini dan *objective function* lainnya.

   Pada pendekatan ini, fungsi evaluasi nilai kubus akan menambahkan nilai 1 setiap kali hasil penjumlahan (baik baris, kolom, tiang, diagonal bidang, dan diagonal ruang) yang tidak sama dengan *magic number*. Sebaliknya, fungsi evaluasi akan menambahkan nilai 0 pada setiap kali penjumlahan memenuhi *magic number*. *Objective function* bertujuan

untuk meminimalisasi hasil evaluasi hingga mencapai 0 (semua evaluasi aturan memenuhi *magic number*). Dalam notasi kasar matematika, fungsi ini dapat ditulis sebagai berikut.

$$(obj.1)\Sigma_{i=1}^{i=109}(eval_i = 315)$$

2. **Menghitung selisih absolut hasil persamaan aturan dengan *magic number***

   Cara ini adalah cara intuitif untuk mengevaluasi seberapa menyimpang suatu keadaan terhadap target. Dengan pendekatan ***cost based***, fungsi evaluasi aturan akan menjumlahkan semua selisih absolut yang ada. Oleh karena itu, fungsi evaluasi akan menambahkan 0 jika susunan memenuhi *magic number*. Sebaliknya, fungsi evaluasi akan menambahkan sejumlah selisih jika susunan tidak memenuhi *magic number*. *Objective function* bertujuan untuk meminimalisasi hasil evaluasi hingga mencapai 0. Dalam notasi kasar matematika, fungsi ini dapat ditulis sebagai berikut.

$$(obj.2)\Sigma_{i=1}^{i=109}(|eval_i - 315|)$$

3. **Menghitung *mean squared error* (MSE) hasil persamaan aturan dengan *magic number***

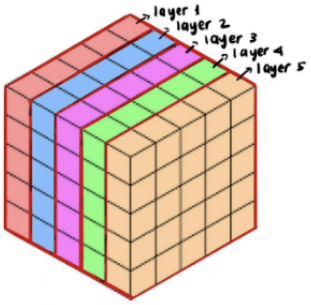
   Cara ini merupakan cara alternatif untuk mengevaluasi seberapa menyimpang suatu keadaan terhadap target. Namun, dengan pendekatan ***cost based***, kubus yang memiliki susunan angka yang lebih jauh dari *magic number* akan mendapat hukuman (*punishment*) akibat adanya operasi kuadrat. Fungsi evaluasi akan menambahkan sejumlah selisih yang dikuadratkan jika susunan tidak memenuhi *magic number*. Oleh karena itu, *objective function* memiliki target yang serupa dengan cara lainnya, yaitu meminimalisasi hasil evaluasi hingga mencapai 0. Dalam notasi kasar matematika, fungsi ini dapat ditulis sebagai berikut.

$$(obj.3)\frac{1}{n} * \Sigma_{i=1}^{i=109}(eval_i - 315)^2$$

Dari pilihan *objective function* di atas, cara yang dipakai adalah cara **pertama** yaitu **menghitung berapa kali *magic number* dipenuhi dalam susunan kubus**. Cara pertama dipilih karena nilai yang dihasilkan dari *local search* dengan *objective function* ini akan memberikan hasil yang

lebih substansial dibandingkan dengan *objective function* lainnya. Hasil *local search* menunjukkan susunan kubus yang hampir memenuhi syarat diagonal *magic cube* yang sudah dijabarkan di atas.

Untuk menentukan berapa kali *magic number* dipenuhi dalam susunan kubus, dapat dihitung terlebih dahulu jumlah rusuk yang dapat memuat *magic number*, yaitu rusuk sebagai baris, kolom, tiang, maupun diagonal ruang dan bidang.



Untuk memudahkan, penggambaran di samping terdapat sebuah kubus 5 x 5 x 5 yang sudah dibagi menjadi 5 *layer*. Berikut pencarian jumlah rusuk dari kubus di samping:

Jumlah baris/kolom/tiang = n^2

- Jumlah rusuk baris + kolom + tiang = 3 x n^2 = 3 x $(5)^2$ = 75
- Jumlah rusuk diagonal bidang = 2 x 5 x 3 = 30
- Jumlah diagonal ruang = 4

Total rusuk yang ada = 75 + 30 + 4 = 109. Terdapat 109 rusuk di dalam kubus 5 x 5 x 5 ini harus memenuhi *magic number* 315 dengan mengacak *random* seluruh angka 1-125 tanpa perulangan.

II. Penjelasan Implementasi Algoritma Local Search

1. Steepest Ascent Hill Climbing

Source Code	Deskripsi
`type SteepestAscent struct {` `  Experiment` `  ActualIteration int` `}`	SteepestAscent adalah sebuah *struct* yang merepresentasikan sebuah *type* dalam menyimpan informasi algoritma optimasi *magic cube*. Di dalam *struct* ini terdapat 2 atribut yaitu atribut *Experiment* yang akan menyimpan setiap seluruh state selama proses pencarian. Terakhir, atribut *actualiteration* menyimpan jumlah setiap iterasi yang dijalankan dalam bentuk int.

`func NewSteepestAscent(c *Cube) *SteepestAscent {` `  sta := &SteepestAscent{}` `  sta.Experiment = *NewExperiment(c.Clone())` `  sta.ActualIteration = 0` `  return sta` `}`	Fungsi NewSteepestAscent digunakan untuk membuat dan menginisialisasi objek untuk SteepestAscent dengan parameter cube. Fungsi ini akan mengembalikan objek SteppestAscent yang siap untuk digunakan dalam pencarian solusi optimal algoritma.
`func (sta *SteepestAscent) Run() {` `  start := time.Now()` `  init := sta.Experiment.GetState(0)` `  current := init.Clone()` `  i := 0` `  for {` `  neighbor := current.FindBestNeighbor()` `  if neighbor.Value >= current.Value {` `  break` `  }` `  current.Copy(neighbor)` `  i++` `  sta.ActualIteration = i` `sta.Experiment.AppendState(current)` `  }` `sta.Experiment.SetRuntime(time.Since(start))` `}`	Fungsi ini bertujuan untuk menemukan solusi optimal untuk suatu masalah dengan menggunakan metode *hill climbing*. Fungsi ini akan mencari *neighbor* terbaik dari kondisi saat ini (current.FindBestNeighbor) dan akan berhenti ketika tidak ada tetangga yang memiliki *neighbor.value* yang lebih baik yaitu *state* yang memiliki nilai fungsi objektif yang lebih rendah daripada kondisi saat ini. Jika neighbor.value nilai tetangga lebih rendah daripada *current.value*, maka *current.value* akan diperbarui dengan neighbor.value terbaik menggunakan *current.copy(neighbor)* *Loop* ini akan terus berjalan hingga tidak ada lagi kondisi neighbor yang lebih baik lagi.

2. Hill-climbing with Sideways Move

Source Code	Deskripsi
`type SidewaysMove struct {` `  Experiment` `  ActualIteration int`	SidewaysMove adalah sebuah *struct* yang merepresentasikan sebuah *type* dalam menyimpan informasi algoritma optimasi

Source Code	Deskripsi
`MaxSideways int` `}`	*magic cube*. Di dalam *struct* ini terdapat 3 atribut yaitu atribut *Experiment* akan menyimpan setiap eksperimen yang mencatat iterasi-iterasi dari semua pencarian. Kedua terdapat atribut *actualiteration* menyimpan jumlah setiap iterasi yang dijalankan dalam bentuk int dan atribut *MaxSideways* akan menetapkan jumlah maksimal dari setiap *sideways moves* yang diizinkan selama proses optimasi dalam bentuk int.
`func NewSidewaysMove(cube *Cube, maxSideways int) *SidewaysMove {` `sm := &SidewaysMove{` `MaxSideways: maxSideways,` `}` `sm.Experiment = *NewExperiment(cube.Clone())` `sm.ActualIteration = 0` `return sm` `}`	Fungsi *NewSidewaysmove* merupakan *constructor* untuk menginisialisasi objek SidewaysMove dengan parameter cube dan maxSideways. Fungsi ini akan menyalin kondisi awal dari cube dan menginisialisasi experiment untuk mencatat setiap langkah pencarian, dan akan menginput atribut *Maxsideways* sebagai batasan jumlah gerakan *sideways mov*e yang diperbolehkan.
`func (sm *SidewaysMove) Run() {` `start := time.Now()` `init := sm.Experiment.GetState(0)` `current := init.Clone()` `i := 0` `sideways := 0` `for {` `neighbor := current.FindBestNeighbor()` `if neighbor.Value > current.Value {` `break` `} else if neighbor.Value == current.Value {` `sideways++` `if sideways > sm.MaxSideways {` `break`	Fungsi run ini digunakan untuk menjalankan proses pencarian solusi optimal dengan beberapa sideways move yang diizinkan. Fungsi ini akan mencari *neighbor* terbaik dari kondisi saat ini (current.FindBestNeighbor) dan akan berhenti ketika tidak ada tetangga yang memiliki *neighbor.value* yang lebih baik yaitu *current.value* , *loop* berhenti, atau karena pencarian telah mencapai solausi optimal atau puncak lokalnya. Jika *neighbor.value* sama dengan *current.value* , gerakan ini dianggap sebagai sideways move dan akan dihitung setiap kondisi ini terjadi. Namun, jika nilai tetangga lebih buruk dari current state maka penghitung sideways akan di reset ke 0. Namun jika nilainya terbaik maka, current akan diperbarui dengan

Source Code	Deskripsi
<pre>} } else { sideways = 0 } current.Copy(neighbor) i++ sm.ActualIteration = i sm.Experiment.AppendState(current) } sm.Experiment.SetRuntime(time.Since(start)) }</pre>	neighbor.value nya.
<pre>func (sm *SidewaysMove) Plot(name string) { p := plot.New() e := sm.Experiment text := "Plot " + name text += fmt.Sprintf("\nFinal State Objective Value: %v", e.State[len(e.State)-1].Value) text += fmt.Sprintf("\nRuntime: %v", e.GetRuntime()) text += fmt.Sprintf("\nMax Sideways: %v", sm.MaxSideways) p.Title.Text = text p.X.Label.Text = "Iteration" p.Y.Label.Text = "Objective Function" p.Add(plotter.NewGrid()) pts := make(plotter.XYs, len(e.State)) for i, cube := range e.State { pts[i].X = float64(i) pts[i].Y = float64(cube.Value)</pre>	Fungsi Plot berfungsi untuk membuat grafik yang menampilkan perkembangan nilai *objective function* selama iterasi dan restart. Fungsi ini akan menggambarkan setiap perubahan nilai *objective function* dari setiap iterasi dan menyimpannya dalam bentuk gambar. Fungsi juga memberikan informasi tambahan seperti Final state Objective value, Runtine dan MaxSideways.

Source Code	Deskripsi
`}` `line, err := plotter.NewLine(pts)` `if err != nil {` `panic(err)` `}` `p.Add(line)` `fileName := "img/" + name + ".png"` `if err := p.Save(8*vg.Inch, 8*vg.Inch, fileName); err != nil {` `panic(err)` `}` `}`	

3. Random Restart Hill-climbing (Steepest)

Source Code	Deskripsi
`type RR_sta struct {` `Restart []SteepestAscent` `MaxRestart int` `ActualRuntime time.Duration` `}`	RR_sta merupakan sebuah *struct* yang digunakan untuk merepresentasikan suatu entitas dalam konteks algoritma optimasi berbasis *Random Restart* dengan metode *Steepest Ascent*. Atribut *Restart* menyimpan *slice* dari tipe data *SteepestAscent*, atribut *MaxRestart* menyimpan jumlah maksimal restart yang diizinkan selama proses optimasi dalam bentuk int, dan atribut *ActualRuntime* menyimpan durasi waktu yang dibutuhkan untuk menyelesaikan seluruh proses optimasi dalam bentuk time.Duration.
`func NewRR_sta(cube *Cube, maxRestart int) *RR_sta {` `rr_sta := &RR_sta{` `MaxRestart: maxRestart,` `}`	Fungsi NewRR_sta digunakan untuk menginisialisasi objek RR_sta dengan parameter cube dan maxRestart. Fungsi ini mengatur jumlah restart maksimal (MaxRestart) dan memulai proses pertama

`rr_sta.Restart = append(make([]SteepestAscent, 0), *NewSteepestAscent(cube))` `return rr_sta` `}`	SteepestAscent yang ditambahkan sebagai elemen awal pada slice Restart.
`func (rr_sta *RR_sta) Run() {` `start := time.Now()` `for i := 0; i < rr_sta.MaxRestart; i++ {` `if i != 0 {` `randomState := NewCube()` `sta := NewSteepestAscent(randomState)` `rr_sta.AppendRestart(sta)` `}` `rr_sta.Restart[i].Run()` `if rr_sta.Restart[i].Experiment.GetEndState().Value == 0 {` `break` `}` `}` `rr_sta.ActualRuntime = time.Since(start)` `}`	Fungsi Run menjalankan algoritma *Random Restart* dengan metode *Steepest Ascent* hingga mencapai batas MaxRestart atau hingga ditemukan solusi optimal (*objective value* = 0). Setiap kali dilakukan restart (kecuali iterasi pertama), objek Steepest Ascent akan dibuat dengan randomState. Fungsi juga digunakan untuk menyimpan waktu eksekusi setelah proses selesai.
`func (rr_sta *RR_sta) Plot(name string) {` `p := plot.New()` `text := "Plot " + name` `text += fmt.Sprintf("\nActual Restart: %v", len(rr_sta.Restart))` `text += fmt.Sprintf("\nAverage Iterations: %v", rr_sta.AverageIterations())` `text += fmt.Sprintf("\nFinal State Objective Value: %v", rr_sta.GetFinalObjectiveValue())` `text += fmt.Sprintf("\nRuntime:`	Fungsi Plot berfungsi untuk membuat grafik yang menampilkan perkembangan nilai *objective function* selama iterasi dan restart. Fungsi ini akan menggambarkan setiap perubahan nilai *objective function* dari setiap iterasi dan menyimpannya dalam bentuk gambar. Fungsi juga memberikan informasi tambahan seperti jumlah restart, rata-rata iterasi, nilai akhir *objective function*, dan waktu eksekusi.

```
%v", rr_sta.GetRuntime())
    p.Title.Text = text
    p.X.Label.Text = "Iteration"
    p.Y.Label.Text = "Objective
Function"
    p.Add(plotter.NewGrid())

    var pts plotter.XYs

    count := 0
    for i, restart := range
rr_sta.Restart {
        for j, state := range
restart.Experiment.State {
            if count > CAP_PLOT {
                break
            }
            count++
            pts = append(pts,
plotter.XY{X:
float64(i*len(restart.Experiment.State
) + j), Y: float64(state.Value)})
        }
    }

    line, err := plotter.NewLine(pts)
    if err != nil {
        panic(err)
    }
    p.Add(line)

    fileName := "img/" + name + ".png"

    if err := p.Save(8*vg.Inch,
8*vg.Inch, fileName); err != nil {
        panic(err)
    }
}
```

```
func (rr_sta *RR_sta)
IterationPlot(name string) {
    p := plot.New()

    text := "Iteration Plot " + name
    p.Title.Text = text
```

Fungsi IterationPlot membuat plot yang menunjukkan jumlah iterasi yang dilakukan pada setiap restart dan menyimpannya dalam bentuk gambar. Fungsi juga memberikan gambaran mengenai jumlah iterasi pada setiap restart

`` p.X.Label.Text = "Restart" p.Y.Label.Text = "Iteration" p.Add(plotter.NewGrid()) limit := CAP_PLOT if len(rr_sta.Restart) < CAP_PLOT { limit = len(rr_sta.Restart) } pts := make(plotter.XYs, limit) for i := 0; i < limit; i++ { pts[i].X = float64(i) pts[i].Y = float64(rr_sta.Restart[i].ActualIterat ion) } line, err := plotter.NewLine(pts) if err != nil { panic(err) } p.Add(line) if err := p.Save(8*vg.Inch, 8*vg.Inch, "img/"+text+".png"); err != nil { panic(err) } } ```	dan membantu dalam analisis konvergensi algoritma.
`` func (rr_sta *RR_sta) Dump(name string) { file, err := os.Create("txt/" + name + ".txt") if err != nil { panic(err) } defer file.Close() count := 0 for _, restart := range rr_sta.Restart { for _, state := range ```	Fungsi Dump akan menyimpan data state dari setiap iterasi ke dalam file teks. Setiap state di flatten dan ditulis ke file txt untuk di analisis lebih lanjut. File ini juga akan berhenti untuk menulis apabila jumlah data yang ditulis telah melebihi batas.

`restart.Experiment.State {` `if count > CAP_DUMP {` `break` `}` `count++` `flattened := state.flatten()` `flatString := ""` `for _, value := range flattened {` `flatString += fmt.Sprintf("%v ", value)` `}` `_, err := file.WriteString(flatString + "\n")` `if err != nil {` `panic(err)` `}` `}` `}` `}`	
`func (rr_sta *RR_sta) GetRuntime() time.Duration {` `return rr_sta.ActualRuntime` `}`	Fungsi GetRuntime akan mengembalikan waktu eksekusi (ActualRuntime) dari seluruh proses optimasi untuk mengukur efisiensi algoritma.
`func (rr_sta *RR_sta) GetEndState() Cube {` `return rr_sta.Restart[len(rr_sta.Restart)-1].Experiment.GetEndState()` `}`	Fungsi GetEndState akan mengembalikan solusi akhir dari optimasi yang dilakukan dengan mengambil state akhir dari iterasi terakhir restart.
`func (rr_sta *RR_sta) GetFinalObjectiveValue() int {` `return rr_sta.Restart[len(rr_sta.Restart)-1].Experiment.GetEndState().Value` `}`	Fungsi GetFinalObjectiveValue akan mengembalikan nilai *objective function* dari state akhir pada iterasi terakhir. Nilai *objective function* ini akan digunakan untuk mengetahui kualitas solusi yang diperoleh setelah semua iterasi dan restart selesai.
`func (rr_sta *RR_sta) AppendRestart(sta *SteepestAscent) {`	Fungsi AppendRestart akan menambahkan objek SteepestAscent baru

`rr_sta.Restart = append(rr_sta.Restart, *sta)` `}`	ke dalam *slice Restart*. Fungsi ini digunakan setiap kali ada restart baru yang dibuat dalam proses optimasi.
`func (rr_sta *RR_sta) AverageIterations() int {` `total := 0` `for _, sta := range rr_sta.Restart {` `total += sta.ActualIteration` `}` `return total / len(rr_sta.Restart)` `}`	Fungsi AverageIterations akan menghitung rata-rata jumlah iterasi yang dilakukan pada setiap restart. Fungsi ini dibuat untuk mengetahui seberapa cepat algoritma dapat mencapai solusi dalam setiap proses restart.

4. **Random Restart Hill-climbing (Sideways)**

Source Code	Deskripsi
`type RR_sm struct {` `Restart []SidewaysMove` `MaxRestart int` `MaxSideways int` `ActualRuntime time.Duration` `}`	RR_sm adalah sebuah struct yang merepresentasikan sebuah *type* dalam konteks algoritma optimasi *magic cube* berbasis algoritma *Random Restart* dengan metode *Hill-climbing with Sideways Move*. Di dalam struct ini ada 4 atribut, yaitu atribut *Restart* yang menyimpan *slice* dari *SidewaysMove*, atribut *MaxRestart* yang menyimpan jumlah maksimal restart yang boleh dilakukan dalam bentuk *integer*, atribut *MaxSideways* yang menyimpan jumlah maksimal gerakan *sideways* yang boleh dilakukan dalam bentuk *integer*, dan atribut *ActualRuntime* yang menyimpan total waktu yang diperlukan untuk menyelesaikan seluruh proses optimasi dalam bentuk *integer*.
`func NewRR_sm(cube *Cube, maxRestart int, maxSideways int) *RR_sm {` `rr_sm := &RR_sm{` `MaxRestart: maxRestart,` `MaxSideways: maxSideways,` `}` `rr_sm.Restart =`	NewRR_sm adalah sebuah fungsi yang digunakan untuk menginisialisasi RR_sm dengan parameter cube, maxRestart, dan maxSideways. Fungsi ini juga akan mengatur nilai *MaxRestart* dan *MaxSideways* serta memulai proses *Sideways* baru yang diinisialisasi dengan cube dan ditambahkan sebagai *slice* ke

`append(make([]SidewaysMove, 0), *NewSidewaysMove(cube, maxSideways))` `return rr_sm` `}`	dalam atribut *Restart*.
`func (rr_sm *RR_sm) Run() {` `start := time.Now()` `for i := 0; i < rr_sm.MaxRestart; i++ {` `if i != 0 {` `randomState := NewCube()` `sm := NewSidewaysMove(randomState, rr_sm.MaxSideways)` `rr_sm.AppendRestart(sm)` `}` `rr_sm.Restart[i].Run()` `if rr_sm.Restart[i].Experiment.GetEndState().Value == 0 {` `break` `}` `}` `rr_sm.ActualRuntime = time.Since(start)` `}`	Fungsi Run akan menjalankan algoritma *Random Restart* dengan metode *Hill-climbing with Sideways Move* hingga jumlah iterasi restart yang telah diatur sebelumnya atau ketika mencapai solusi optimal (*objective value* = 0). Setiap proses restart akan dimulai dengan sebuah *randomState* baru (kecuali iterasi pertama) dan menambahkan *SidewaysMove* baru. Fungsi ini juga menyimpan *ActualRuntime* (total waktu keseluruhan proses *restart*).
`func (rr_sm *RR_sm) Plot(name string) {` `p := plot.New()` `text := "Plot " + name` `text += fmt.Sprintf("\nActual Restart: %v", len(rr_sm.Restart))` `text += fmt.Sprintf("\nAverage Iterations: %v", rr_sm.AverageIterations())`	Fungsi Plot digunakan untuk menampilkan perkembangan nilai *objective function* setiap restart selama iterasi berlangsung. Fungsi ini juga akan menampilkan informasi seperti jumlah *restart*, rata-rata iterasi, nilai *objective function* terakhir, total *runtime*, dan jumlah maksimal *sideways*.

```
    text += fmt.Sprintf("\nFinal State
Objective Value: %v",
rr_sm.GetFinalObjectiveValue())
    text += fmt.Sprintf("\nRuntime:
%v", rr_sm.GetRuntime())
    text += fmt.Sprintf("\nMax
Sideways: %v", rr_sm.MaxSideways)
    p.Title.Text = text
    p.X.Label.Text = "Iteration"
    p.Y.Label.Text = "Objective
Function"
    p.Add(plotter.NewGrid())

    var pts plotter.XYs

    count := 0
    for i, restart := range
rr_sm.Restart {
        for j, state := range
restart.Experiment.State {
            if count >= CAP_PLOT {
                break
            }
            count++
            pts = append(pts,
plotter.XY{X:
float64(i*len(restart.Experiment.State
) + j), Y: float64(state.Value)})
        }
    }

    line, err := plotter.NewLine(pts)
    if err != nil {
        panic(err)
    }
    p.Add(line)

    fileName := "img/" + name + ".png"

    if err := p.Save(8*vg.Inch,
8*vg.Inch, fileName); err != nil {
```

`panic(err)` `}` `}`	
`func (rr_sm *RR_sm) IterationPlot(name string) {` `p := plot.New()` `text := "Iteration Plot " + name` `p.Title.Text = text` `p.X.Label.Text = "Restart"` `p.Y.Label.Text = "Iteration"` `p.Add(plotter.NewGrid())` `limit := CAP_PLOT` `if len(rr_sm.Restart) < CAP_PLOT {` `limit = len(rr_sm.Restart)` `}` `pts := make(plotter.XYs, limit)` `for i := 0; i < limit; i++ {` `pts[i].X = float64(i)` `pts[i].Y = float64(rr_sm.Restart[i].ActualIteration)` `}` `line, err := plotter.NewLine(pts)` `if err != nil {` `panic(err)` `}` `p.Add(line)` `if err := p.Save(8*vg.Inch, 8*vg.Inch, "img/"+text+".png"); err != nil {` `panic(err)` `}` `}`	Fungsi IterationPlot akan membuat plot yang menunjukkan jumlah iterasi yang dilakukan pada setiap restart dan disimpan dalam bentuk gambar dan membantu dalam analisis konvergensi algoritma.
`func (rr_sm *RR_sm) Dump(name string)`	Fungsi Dump akan digunakan untuk

<pre>{ file, err := os.Create("txt/" + name + ".txt") if err != nil { panic(err) } defer file.Close() count := 0 for _, restart := range rr_sm.Restart { for _, state := range restart.Experiment.State { if count >= CAP_DUMP { break } count++ flattened := state.flatten() flatString := "" for _, value := range flattened { flatString += fmt.Sprintf("%v ", value) } _, err := file.WriteString(flatString + "\n") if err != nil { panic(err) } } } }</pre>	menyimpan data *state* dari setiap iterasi ke dalam *file* txt. Setiap state di *flatten* dan ditulis ke *file* txt untuk di analisis lebih lanjut. Fungsi ini akan berhenti ketika jumlah data yang ditulis telah melebihi batas.
<pre>func (rr_sm *RR_sm) GetRuntime() time.Duration { return rr_sm.ActualRuntime }</pre>	Fungsi GetRuntime akan mengembalikan total waktu yang diperlukan untuk menjalankan keseluruhan algoritma.
<pre>func (rr_sm *RR_sm) GetEndState() Cube</pre>	Fungsi GetEndState akan mengembalikan

`{` `  return rr_sm.Restart[len(rr_sm.Restart)-1].Experiment.GetEndState()` `}`	*state* terakhir dari proses *restart sideways move* yang terakhir.
`func (rr_sm *RR_sm) GetFinalObjectiveValue() int {` `  return rr_sm.Restart[len(rr_sm.Restart)-1].Experiment.GetEndState().Value` `}`	Fungsi GetFinalObjectiveValue akan mengembalikan nilai *objective value* dari proses *restart sideways move* yang terakhir untuk menunjukkan seberapa dekat *state* terakhir dengan solusi optimal.
`func (rr_sm *RR_sm) AppendRestart(sm *SidewaysMove) {` `  rr_sm.Restart = append(rr_sm.Restart, *sm)` `}`	Fungsi AppendRestart akan menambahkan *slice sideways move* baru ke dalam atribut *Restart*.
`func (rr_sm *RR_sm) AverageIterations() int {` `  total := 0` `  for _, sm := range rr_sm.Restart {` `  total += sm.ActualIteration` `  }` `  return total / len(rr_sm.Restart)` `}`	Fungsi AverageIterations akan mengembalikan rata-rata iterasi pada setiap restart yang dilakukan di keseluruhan algoritma.

5. **Stochastic Hill-climbing**

Source Code	Deskripsi
`type Stochastic struct {` `  Experiment` `  MaxIterations int` `}`	Stochastic adalah sebuah struct yang merepresentasikan sebuah *type* dalam konteks algoritma optimasi *magic cube* berbasis algoritma *Stochastic Hill-climbing*. Di dalam struct ini ada 2 atribut, yaitu atribut *Experiment* untuk memberikan akses ke *method Experiment* dan atribut *MaxIterations* yang menyimpan jumlah maksimal iterasi yang boleh dilakukan dalam bentuk *integer*.

`func NewStochastic(cube *Cube, maxIterations int) *Stochastic {` `  s := &Stochastic{` `  MaxIterations: maxIterations,` `  }` `  s.Experiment = *NewExperiment(cube)` `  return s` `}`	NewStochastic adalah sebuah fungsi yang digunakan untuk menginisialisasi Stochastic dengan parameter cube dan MaxIteration. Fungsi ini akan mengatur nilai *MaxIteration* dan menginisialisasi status awal *experiment* dengan cube yang diberikan.
`func (s *Stochastic) Run() {` `  start := time.Now()` `  init := s.Experiment.GetState(0)` `  current := init.Clone()` `  neighbor := current.Clone()` `  for i := 0; i < s.MaxIterations; i++ {` `  neighbor.Copy(current)` `  neighbor.FindRandomNeighbor()` `  if neighbor.Value < current.Value {` `  current.Copy(neighbor)` `  }` `s.Experiment.AppendState(current)` `  }` `s.Experiment.SetRuntime(time.Since(start))` `}`	Fungsi Run akan menjalankan algoritma Stochastic Hill-climbing dengan membandingkan *current* dan *random neighbour* hingga jumlah maksimal iterasi yang telah diatur sebelumnya dan menambahkan c*urrent state* ke dalam list *Experiment* serta mencatat total *runtime Experiment*.

6. Simulated Annealing

Source Code	Deskripsi
`type SimulatedAnnealing struct {` `  Experiment` `  T  float64` `  InitialT  float64`	SimulatedAnnealing merupakan sebuah *struct* yang digunakan untuk merepresentasikan suatu entitas dalam konteks algoritma optimasi berbasis

`` Boltzmann []float64 stuck int ActualIteration int } ```	*Simulated Annealing*. Atribut *Experiment* untuk memberikan akses ke *method Experiment*, atribut T menyimpan nilai temperatur saat ini dalam tipe float64, atribut *InitialT* menyimpan nilai temperatur awal yang diberikan saat membuat objek dengan tipe float64, atribut *Boltzmann* dengan tipe *slice* akan menyimpan nilai probabilitas penerimaan solusi buruk pada setiap iterasi, atribut *stuck* akan menghitung jumlah iterasi di mana solusi yang lebih buruk diterima berdasarkan probabilitas *Boltzmann* dengan tipe int, dan atribut ActualIteration bertipe int yang menyimpan jumlah iterasi yang sebenarnya dilakukan sebelum algoritma berhenti.
`` func NewSimulatedAnnealing(cube *Cube, initialT float64) *SimulatedAnnealing { sa := &SimulatedAnnealing{} sa.Experiment = *NewExperiment(cube) sa.Boltzmann = make([]float64, 0) sa.stuck = 0 sa.InitialT = initialT return sa } ```	Fungsi NewSimulatedAnnealing merupakan *constructor* untuk membuat objek *SimulatedAnnealing*. Fungsi akan menginisialisasi eksperimen dengan state awal Cube, mengatur temperatur awal (InitialIT), dan menginisialisasi *slice Boltzmann* serta mengembalikan *pointer* ke objek *SimulatedAnnealing* yang baru.
`` func (sa *SimulatedAnnealing) Run() { start := time.Now() init := sa.Experiment.GetState(0) current := init.Clone() neighbor := current.Clone() for i := 1; i < SA_MAX; i++ { sa.schedule(i) sa.ActualIteration = i if sa.T <= CAP_T { // very close to 0 since the T will never touch 0 break ```	Fungsi Run adalah fungsi yang menjalankan Simulated Annealing hingga jumlah iterasi maksimal tercapai (SA_MAX) atau temperatur mendekati nol (CAP_T). Pada setiap iterasi, fungsi akan. - Mengupdate temperatur menggunakan metode *schedule* - Mencari *neighbor random* dan menghitung perubahan nilai - Menghitung probabilitas penerimaan solusi yang lebih buruk menggunakan fungsi *probability* - Jika nilai yang didapatkan lebih baik atau probabilitas lebih besar

<pre>} neighbor.Copy(current) neighbor.FindRandomNeighbor() delta := neighbor.Value - current.Value probability := sa.probability(float64(delta)) random := rand.Float64() sa.AppendProbability(probability) if delta <= 0 { current.Copy(neighbor) sa.Boltzmann[len(sa.Boltzmann)-1] = sa.InitialT } else if probability > random { current.Copy(neighbor) sa.stuck++ } sa.Experiment.AppendState(current) } sa.Experiment.SetRuntime(time.Since(start)) }</pre>	dibandingkan angka *random*, maka nilai akan diperbarui - Menambahkan *state* saat ini ke eksperimen untuk pelacakan Fungsi ini juga akan mengukur waktu eksekusi dan menyimpannya di SetRuntime.
<pre>func (sa *SimulatedAnnealing) schedule(t int) { sa.T = sa.InitialT * math.Pow(COOLING_RATE, float64(t)) }</pre>	Fungsi schedule adalah mengatur temperatur menggunakan *cooling schedule* berdasarkan iterasi ke-t. *Cooling rate* adalah konstanta yang menentukan seberapa cepat temperatur menurun.
<pre>func (sa *SimulatedAnnealing) probability(delta float64) float64 { return math.Exp(-delta / sa.T) }</pre>	Fungsi probability adalah fungsi yang menghitung probabilitas penerimaan solusi yang buruk menggunakan fungsi *Boltzmann*: $P = e^{-delta/T}$. Delta adalah selisih nilai *objective function* antara *state* saat ini dan *state neighbor*. Nilai probabilitas akan

	digunakan untuk menentukan apakah solusi yang lebih buruk akan diterima berdasarkan suhu saat ini.
`` func (sa *SimulatedAnnealing) Plot(name string) { p := plot.New() e := sa.Experiment text := "Plot " + name text += fmt.Sprintf("\nIteration: %v", len(e.State)) text += fmt.Sprintf("\nFinal State Objective Value: %v", e.State[len(e.State)-1].Value) text += fmt.Sprintf("\nRuntime: %v", e.GetRuntime()) text += fmt.Sprintf("\nStuck: %v", sa.stuck) p.Title.Text = text p.X.Label.Text = "Iteration" p.Y.Label.Text = "Objective Function" p.Add(plotter.NewGrid()) limit := CAP_PLOT if len(e.State) < CAP_PLOT { limit = len(e.State) } pts := make(plotter.XYs, limit) for i := 0; i < limit; i++ { pts[i].X = float64(i) pts[i].Y = float64(cube.Value) } line, err := plotter.NewLine(pts) if err != nil { panic(err) } p.Add(line) fileName := "img/" + name + ".png" ```	Fungsi Plot adalah fungsi untuk membuat plot perkembangan nilai *objective function* selama iterasi dan menyimpan dalam bentuk gambar. Fungsi menggunakan data *Experiment* untuk membuat grafik yang menunjukkan perubahann nilai *objective function* pada setiap iterasi. Fungsi ini akan memberikan informasi tambahan seperti jumlah iterasi, nilai *objective function* terakhir, *runtime*, dan jumlah *stuck*.

`` if err := p.Save(8*vg.Inch, 8*vg.Inch, fileName); err != nil { panic(err) } } ```	
`` func (sa *SimulatedAnnealing) BoltzmannPlot(name string) { p := plot.New() text := "Boltzmann Plot " + name p.Title.Text = text p.X.Label.Text = "Iteration" p.Y.Label.Text = "Boltzmann Function" p.Add(plotter.NewGrid()) limit := CAP_BOLTZMANN_PLOT if len(sa.Boltzmann) < CAP_BOLTZMANN_PLOT { limit = len(sa.Boltzmann) } pts := make(plotter.XYs, limit) for i := 0; i < limit; i++ { pts[i].X = float64(i) pts[i].Y = float64(sa.Boltzmann[i]) } scatter, err := plotter.NewScatter(pts) if err != nil { panic(err) } p.Add(scatter) if err := p.Save(8*vg.Inch, 8*vg.Inch, "img/"+text+".png"); err != nil { panic(err) } } ```	Fungsi BoltzmannPlot adalah fungsi untuk membuat plot yang menampilkan perubahan nilai probabilitas penerimaan solusi lebih buruk selama iterasi dan menyimpan gambar. Fungsi ini juga digunakan untuk menggambarkan perubahan probabilitas seiring dengan penurunan temperatur.

`func (sa *SimulatedAnnealing) AppendProbability(probability float64) {` `  sa.Boltzmann = append(sa.Boltzmann, probability)` `}`	Fungsi AppendProbability adalah fungsi untuk menambahkan nilai probabilitas yang telah dihitung ke dalam *slice Boltzmann*. Fungsi ini digunakan untuk mencatat setiap nilai probabilitas yang dihasilkan selama iterasi.

7. Genetic Algorithm

Source Code	Deskripsi
`type GeneticAlgorithm struct {` `  Experiment` `  PopulationSize int` `  Population []Cube` `  MaxIterations int` `  ActualIteration int` `  AvgObjective []int` `}`	GeneticAlgorithm adalah sebuah struct yang merepresentasikan sebuah type dalam konteks algoritma optimasi *magic cube* berbasis algoritma *Genetic Algorithm*. Di dalam struct ini ada 6 atribut, yaitu atribut *Experiment* untuk memberikan akses ke *method Experiment*, atribut *PopulationSize* yang menyimpan jumlah populasi (*cube*) di setiap generasi, atribut *Population* yang menyimpan semua *cube* yang ada di *current generation*, atribut *MaxIterations* yang menyimpan jumlah maksimal iterasi yang boleh dilakukan dalam bentuk *integer*, atribut *ActualIteration* yang menyimpan nomor *current generation* saat algoritma sedang berjalan, dan atribut *AvgObjective* yang akan menyimpan rata-rata *objective value* dari setiap generasi.
`func NewGeneticAlgorithm(cube *Cube, populationSize int, maxIterations int) *GeneticAlgorithm {` `  ga := &GeneticAlgorithm{}` `  ga.PopulationSize = populationSize` `  ga.MaxIterations = maxIterations` `  ga.ActualIteration = 0` `  ga.AvgObjective = []int{}` `  ga.Population = make([]Cube, ga.PopulationSize)`	NewGeneticAlgorithm adalah sebuah fungsi untuk menginisialisasi GeneticAlgorithm dengan parameter cube, *populationSize*, dan *MaxIteration*. Fungsi ini mengatur nilai *PopulationSize* dan *MaxIteration* serta membuat sebuah klon dari input *cube* dan *randomized cubes* untuk memenuhi jumlah populasi.

```
    ga.Population[0] = *cube.Clone()
    for i := 1; i < ga.PopulationSize;
i++ {
        ga.Population[i] = *NewCube()
    }

    ga.Experiment =
*NewExperiment(cube.Clone())

    elitePercent = BIG_ELITE
    if populationSize < 200 {
        elitePercent = SMALL_ELITE
    }

    eliteDeath = int(ELITE_SPAN *
float64(maxIterations))
    eliteSize =
int(float64(populationSize) *
elitePercent)

    return ga
}
```

```
func (ga *GeneticAlgorithm) Run() {
    start := time.Now()
    eliteAge = 0
    eliteSize =
int(float64(ga.PopulationSize) *
elitePercent)

    for {
        ga.Sort()

        // add best cube to the
experiment state
        if ga.ActualIteration == 0 {
            ga.Experiment.State[0] =
ga.Population[0]
        } else {
```

Fungsi Run akan menjalankan *loop* utama dari algoritma *Genetic Algorithm*, termasuk melakukan *sorting* pada populasi berdasarkan *objective value*, menambahkan *cube* terbaik ke dalam *Experiment state*, serta menghitung rata-rata *objective values* hingga berhenti pada kondisi yang diinginkan (*objective value* = 0 atau mencapai jumlah iterasi maksimum).

`` ga.Experiment.AppendState(&ga.Populati on[0]) } // calculate average objective value avg := 0 for i := 0; i < ga.PopulationSize; i++ { avg += ga.Population[i].Value } avg /= ga.PopulationSize ga.AvgObjective = append(ga.AvgObjective, avg) // break condition if ga.Population[0].Value == 0 || ga.EndSearch() { break } ga.NextGeneration() } ga.Experiment.SetRuntime(time.Since(st art)) } ```	
`` func (ga *GeneticAlgorithm) Sort() { for i := 0; i < ga.PopulationSize; i++ { for j := i + 1; j < ga.PopulationSize; j++ { if ga.Population[i].Value > ga.Population[j].Value { ga.Population[i], ga.Population[j] = ga.Population[j], ga.Population[i] } ```	Fungsi Sort akan melakukan sorting secara *ascend* (naik) berdasarkan *objective value* yang dimiliki oleh masing-masing *cube*.

`` } } } ```	
`` func (ga *GeneticAlgorithm) NextGeneration() { ga.ActualIteration++ nextPopulation := make([]Cube, 0) if eliteAge < eliteDeath { eliteAge++ nextPopulation = append(nextPopulation, ga.Population[:eliteSize]...) } else { eliteAge = 0 } // selection wheel selectionWheel := ga.CreateSelectionWheel() for { if len(nextPopulation) == ga.PopulationSize { break } // selection parent1, parent2 := ga.SelectParents(selectionWheel) // crossover child1, child2 := ga.Crossover(parent1, parent2) // mutation ga.Mutate(child1) ga.Mutate(child2) if !ga.IsDuplicate(*child1) && !ga.IsDuplicate(*child2) { nextPopulation = ```	Fungsi NextGeneration akan digunakan untuk membuat generasi baru dengan cara mempertahankan "*elite individual*", melakukan pemilihan *parent*, melakukan *crossover*, dan melakukan mutasi untuk menghasilkan populasi baru.

`append(nextPopulation, *child1)` `}` `if !ga.IsDuplicate(*child2) && len(nextPopulation) < ga.PopulationSize {` `nextPopulation = append(nextPopulation, *child2)` `}` `}` `ga.Population = nextPopulation` `}`	
`func (ga *GeneticAlgorithm) CreateSelectionWheel() []float64 {` `selectionWheel := make([]float64, 0)` `sum := 0` `for i := 0; i < len(ga.Population); i++ {` `sum += ga.Population[i].Value` `}` `position := 0.0` `for i := 0; i < len(ga.Population); i++ {` `position += float64(ga.Population[i].Value) / float64(sum)` `selectionWheel = append(selectionWheel, position)` `}` `return selectionWheel` `}`	Fungsi CreateSelectionWheel akan membuat sebuah probability wheel berdasarkan *objective value* dan nantinya akan digunakan untuk proses pemilihan *parent*.
`func (ga *GeneticAlgorithm) SelectParents(selectionWheel []float64) (Cube, Cube) {` `parent1 := ga.Population[0]`	Fungsi SelectParents dengan parameter *selectionWheel* akan digunakan untuk memilih dua *parent* yang berbeda dari *probability wheel* secara random.

```
    parent2 := ga.Population[0]

    prob1 := rand.Float64()
    for i := 0; i <
len(selectionWheel); i++ {
        if prob1 <= selectionWheel[i] {
            parent1 = ga.Population[i]
            break
        }
    }

    for {
        prob2 := rand.Float64()
        for i := 0; i <
len(selectionWheel); i++ {
            if prob2 <=
selectionWheel[i] {
                parent2 =
ga.Population[i]
                break
            }
        }
        if parent1 != parent2 {
            break
        }
    }

    return parent1, parent2
}
```

```
func (ga *GeneticAlgorithm)
Crossover(parent1 Cube, parent2 Cube)
(*Cube, *Cube) {
    parent1 = *parent1.Clone()
    parent2 = *parent2.Clone()
    parentFlat1 := parent1.flatten()
    parentFlat2 := parent2.flatten()

    child1 := NewCube()
    child2 := NewCube()
```

Fungsi Crossover dengan parameter *parent1* dan *parent2* akan digunakan untuk melakukan *crossover* antara dua *parent* untuk menghasilkan dua populasi baru dengan cara mengcopy sebagian gen dari salah satu parent dan mengisi sisanya dengan gen dari parent lain. Crossover yang digunakan adalah order crossover yang cocok untuk permutasi.

```
    // order crossover (ox1)
    var point1, point2 int
    for {
        point1 = rand.Intn(ELEMENT)
        point2 = rand.Intn(ELEMENT)
        if point1 != point2 {
            if point1 > point2 {
                point1, point2 =
point2, point1
            }
            break
        }
    }

    childFlat1 := make([]uint8,
ELEMENT)
    childFlat2 := make([]uint8,
ELEMENT)

    // initialize child with 0 to
indicate empty slots
    for i := 0; i < ELEMENT; i++ {
        childFlat1[i] = uint8(0)
        childFlat2[i] = uint8(0)
    }

    // copy the genes within points
range from respective parents
    for i := point1; i <= point2; i++ {
        childFlat1[i] = parentFlat1[i]
        childFlat2[i] = parentFlat2[i]
    }

    // get the list of the other
parent's genes that are not in the
points range
    remainingGenes1 := []uint8{}
    remainingGenes2 := []uint8{}
    for i := 0; i < ELEMENT; i++ {
        if !contains(childFlat1,
parentFlat2[i]) {
```

<pre>remainingGenes1 = append(remainingGenes1, parentFlat2[i]) } if !contains(childFlat2, parentFlat1[i]) { remainingGenes2 = append(remainingGenes2, parentFlat1[i]) } } // fill the remaining genes in child for i := 0; i < ELEMENT; i++ { if childFlat1[i] == 0 { childFlat1[i] = remainingGenes1[0] remainingGenes1 = remainingGenes1[1:] } if childFlat2[i] == 0 { childFlat2[i] = remainingGenes2[0] remainingGenes2 = remainingGenes2[1:] } } child1.unflatten(childFlat1) child2.unflatten(childFlat2) return child1, child2 }</pre>	
<pre>func contains(arr []uint8, val uint8) bool { for _, v := range arr { if v == val { return true }</pre>	Fungsi contains dengan parameter []uint8 dan uint8 akan digunakan untuk mengecek apakah ada *specific value* di dalam *slice* uint8.

`` } return false } ```	
`` func (ga *GeneticAlgorithm) Mutate(cube *Cube) { if ga.IsGettingConvergen() { mutationProb = BIG_MUTATION } else { mutationProb = SMALL_MUTATION } if rand.Float64() >= mutationProb { return } cubeFlat := cube.flatten() // inversion i := rand.Intn(len(cubeFlat)) j := i for i == j { d := rand.Float64() * float64(len(cubeFlat)-1) j = (i + int(d)) % len(cubeFlat) } // keep track of the selected genes to invert selectedGenes := make([]uint8, 0) if i <= j { selectedGenes = append(selectedGenes, cubeFlat[i:j+1]...) } else { selectedGenes = append(selectedGenes, cubeFlat[i:]...) selectedGenes = append(selectedGenes, cubeFlat[:j+1]...) ```	Fungsi Mutate dengan parameter *cube* akan digunakan untuk melakukan mutasi pada *cube* untuk mendapatkan individu yang benar-benar unik menggunakan inversi untuk memodifikasi gen. Mutate dilakukan dengan menggunakan inversion yang cocok untuk permutasi.

`}` `// invert the selected genes` `for k := 0; k < len(selectedGenes)/2; k++ {` `selectedGenes[k], selectedGenes[len(selectedGenes)-1-k] = selectedGenes[len(selectedGenes)-1-k], selectedGenes[k]` `}` `if i < j {` `for k := i; k <= j; k++ {` `cubeFlat[k] = selectedGenes[k-i]` `}` `} else {` `for k := i; k < len(cubeFlat); k++ {` `cubeFlat[k] = selectedGenes[k-i]` `}` `for k := 0; k <= j; k++ {` `cubeFlat[k] = selectedGenes[len(cubeFlat)-i+k]` `}` `}` `cube.unflatten(cubeFlat)` `}`	
`func (ga *GeneticAlgorithm) IsGettingConvergen() bool {` `if ga.ActualIteration < 50 {` `return false` `}` `for i := 1; i < 50; i++ {` `if ga.AvgObjective[ga.ActualIteration-i]`	Fungsi IsGettingConvergen akan digunakan untuk mengecek apakah rata-rata *objective value* tetap stabil selama 50 iterasi terakhir.

`!= ga.AvgObjective[ga.ActualIteration-1] { return false } } return true }`	
`func (ga *GeneticAlgorithm) EndSearch() bool { return ga.ActualIteration >= ga.MaxIterations }`	Fungsi EndSearch akan digunakan untuk mengecek apakah algoritma sudah mencapai jumlah iterasi maksimum atau belum.
`func (ga *GeneticAlgorithm) IsDuplicate(cube Cube) bool { for i := 0; i < len(ga.Population); i++ { if cube.IsSame(&ga.Population[i]) { return true } } return false }`	Fungsi IsDuplicate dengan parameter *cube* akan digunakan untuk mengecek apakah sebuah *cube* sudah ada di dalam *current population* untuk menghindari duplikat.
`func (ga *GeneticAlgorithm) Plot(name string) { p := plot.New() e := &ga.Experiment text := "Plot " + name text += fmt.Sprintf("\nIteration: %v", ga.ActualIteration) text += fmt.Sprintf("\nFinal State Objective Value: %v", e.State[len(e.State)-1].Value) text += fmt.Sprintf("\nRuntime: %v", e.GetRuntime())`	Fungsi Plot adalah fungsi untuk membuat plot yang menampilkan perkembangan nilai *objective function* selama iterasi dan disimpan dalam bentuk gambar. Fungsi ini juga akan menampilkan informasi seperti jumlah iterasi, nilai *objective function* terakhir, total runtime, ukuran populasi, jumlah maksimal iterasi, dan *objective function*.

```
    text += fmt.Sprintf("\nPopulation
Size: %v", ga.PopulationSize)
    text += fmt.Sprintf("\nMax
Iterations: %v", ga.MaxIterations)
    p.Title.Text = text
    p.X.Label.Text = "Iteration"
    p.Y.Label.Text = "Objective
Function"
    p.Add(plotter.NewGrid())

    limit := CAP_DUMP
    if len(e.State) < CAP_DUMP {
        limit = len(e.State)
    }

    pts := make(plotter.XYs, limit)
    avgPts := make(plotter.XYs, limit)

    for i := 0; i < limit; i++ {
        pts[i].X = float64(i)
        pts[i].Y =
float64(e.State[i].Value)
        avgPts[i].X = float64(i)
        avgPts[i].Y =
float64(ga.AvgObjective[i])
    }

    line, err := plotter.NewLine(pts)
    if err != nil {
        panic(err)
    }
    p.Add(line)

    avgLine, err :=
plotter.NewLine(avgPts)
    if err != nil {
        panic(err)
    }
    avgLine.LineStyle.Color =
plotutil.Color(0)
    p.Add(avgLine)
```

```
    p.Legend.Add("Best", line)
    p.Legend.Add("Average", avgLine)

    fileName := "img/" + name + ".png"

    if err := p.Save(8*vg.Inch,
8*vg.Inch, fileName); err != nil {
        panic(err)
    }
}
```

III. Hasil Eksperimen dan Analisis

1. Steepest Ascent Hill-climbing

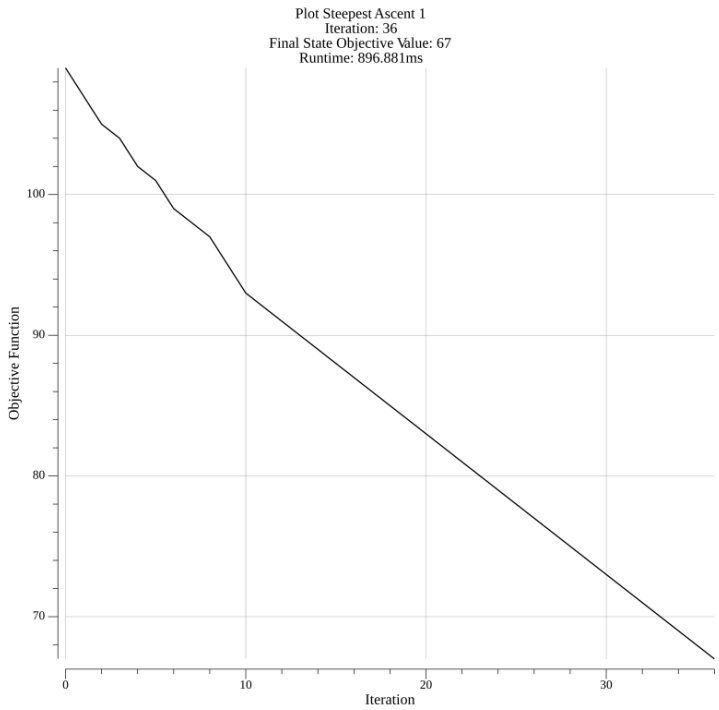
a. Hasil Eksperimen

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

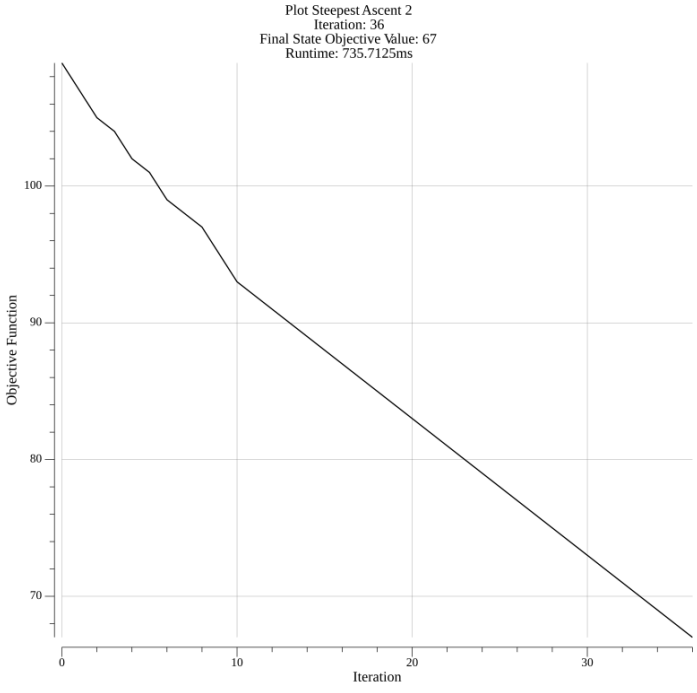
ALGORITHM	RUN	TIME	VALUE	V1	V2	ITERATION	STUCK	POPULATION
Initial			109	7786	757748			
Steepest Ascent	1	0.90	67	4557	462347	36		
Steepest Ascent	2	0.74	67	4557	462347	36		
Steepest Ascent	3	0.73	67	4557	462347	36		

```
Generating dump file...
Generating plot file...
```

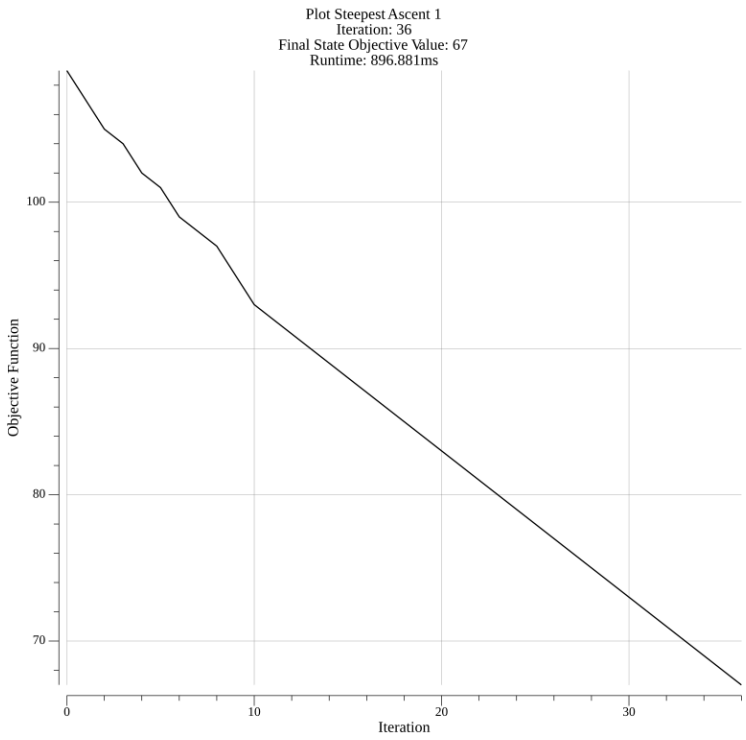
Percobaan 1



Percobaan 2



Percobaan 3



b. Analisis Hasil

Algoritma Steepest Ascent Hill Climbing memiliki kelemahan untuk rentan terjebak di *local optima.* Pada 3 percobaan yang dilakukan Algoritma ini mencapai nilai akhir 67 yang jauh lebih rendah dibanding initial value yaitu 109. Steepest Ascent melakukan pencarian lokal yang cenderung terjebak di local optima karena ia mencari perubahan kecil tanpa ada teknis untuk keluar dari lembah lokal tersebut. Durasi waktu pencarian berkisar antara 0.73 hingga 0.90 detik yang menunjukkan algoritma ini sangat efisien dalam pemrosesannya.

Pada setiap percobaan yang dilakukan algoritma ini menghasilkan nilai akhir yang sama yaitu 67 dan nilai V1 seta V2 yang identik pada setiap percobaannya. Konsistensi ini menunjukkan bahwa algoritma steepest ascent selalu konvergen pada solusi yang sama setiap kali dijalankan sehingga ada kemungkinan berada pada local optima yang sama.

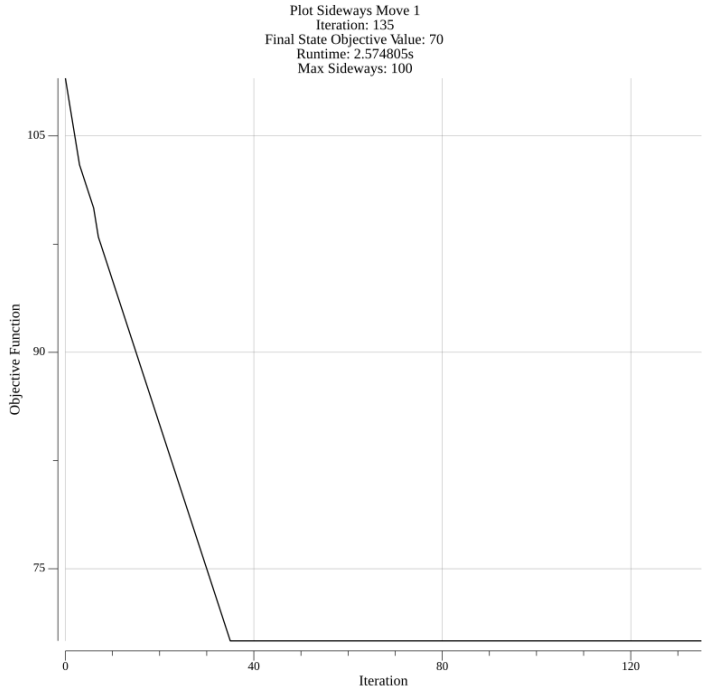
2. Hill-climbing with Sideways Move

a. Hasil Eksperimen

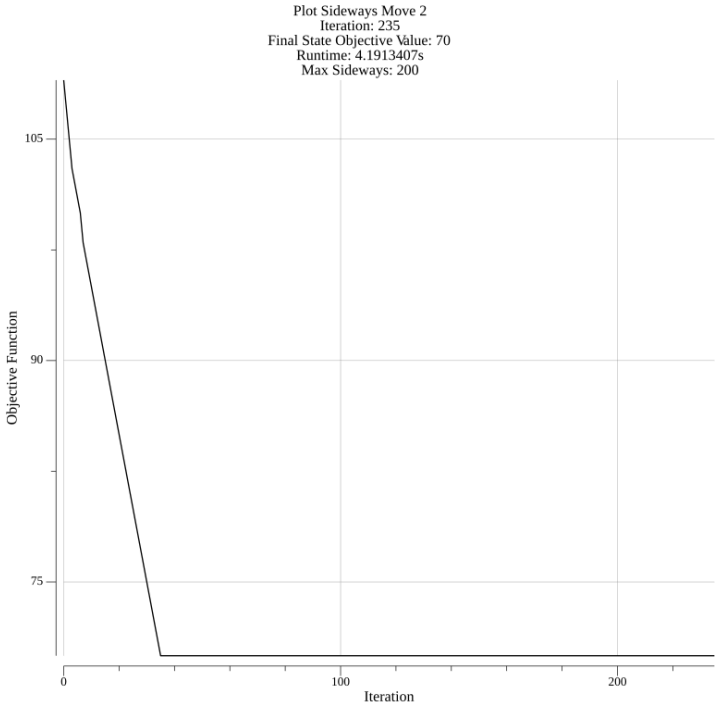
```
Running sideways move...
Initial state:
 41  89  67  65   2     57  80  84  52  70     75  96  72 117   6    108  29  14 125  21     87 109   4  23 120
  5 104  33  43  59     90  74  38  50  97     64  17  24 123  79     37 116  44 113 101     39   1  47  46  18
 15  58  99  22  83    105  73  28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

```
=============================================================================================================
ALGORITHM               RUN     TIME    VALUE   V1      V2      ITERATION       STUCK   POPULATION
=============================================================================================================
Initial                                 109     7520    736618
Sideways Move           1       2.57    70      5214    531302  135
Sideways Move           2       4.19    70      5214    531302  235
Sideways Move           3       6.11    70      5214    531302  335
Generating dump file...
Generating plot file...
=============================================================================================================
```

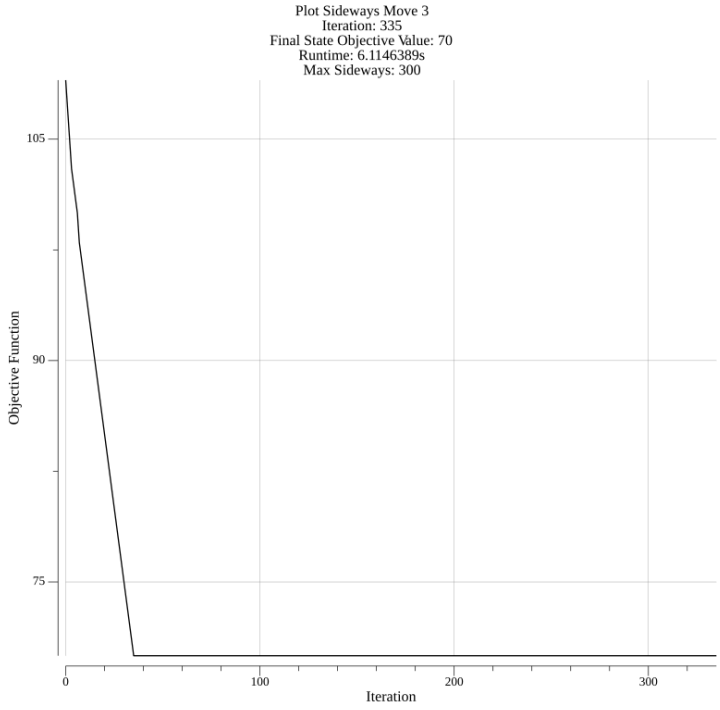
Percobaan 1 (Max Sideways = 100)



Percobaan 2 (Max Sideways = 200)



Percobaan 3 (Max Sideways = 300)



b. Analisis Hasil

Algoritma Hill-climbing with Sideways Move memiliki kelemahan yang sama dengan Steepest Ascent Hill-climbing yaitu rentan terjebak di local optima. Namun, Sideways Move ini memiliki sedikit flexibilitas daripada Steepest Ascent karena mengizinkan pergerakan ke nilai yang sama. Jika dilihat dari nilai yang dihasilkan oleh kedua algoritma ini, tidak ada perubahan yang signifikan dan juga algoritma Sideways Move bisa jadi terjebak di plateau atau dataran yang menyebabkan konvergensi di optimum lokal.

Masih sama seperti *steepest ascent* algoritma ini masih belum optimal karena belum ada teknis untuk keluar dari local optima karena cenderung bertahan pada nilai yang sama dengan setiap percobaan dari batasan sideways move 100, 200 dan 300 memberikan nilai yang konsisten di angka 70. Waktu pencarian untuk algoritma ini bervariasi di rentang 2.57s hingga 6.11s bergantung pada jumlah iterasi yang dilakukan. Semakin banyak iterasinya semakin lama juga waktu yang dibutuhkan.

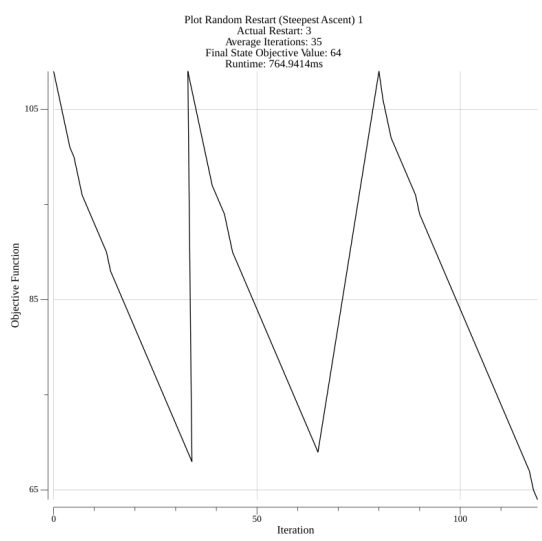
3. Random Restart Hill-climbing (Steepest)

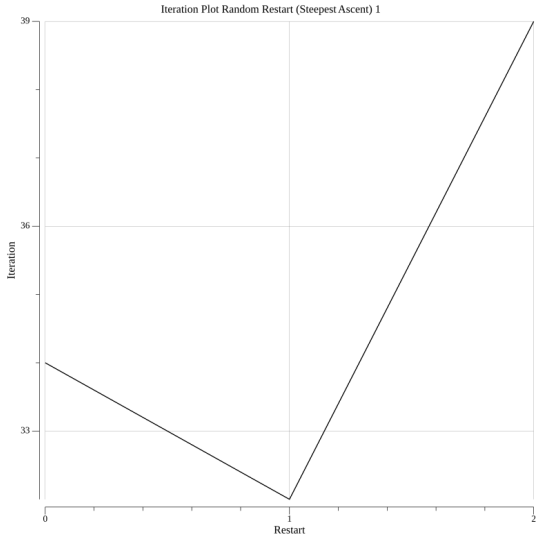
a. Hasil Eksperimen

```
====================================RANDOM RESTART (STEEPEST ASCENT)====================================
Enter max restart for rr_sta1: 3
Enter max restart for rr_sta2: 5
Enter max restart for rr_sta3: 7
Running random restart (steepest ascent)...
Running iteration 1 of random restart (steepest ascent)...
Running iteration 2 of random restart (steepest ascent)...
Running iteration 3 of random restart (steepest ascent)...
Initial state:
109  24  53  86 120      38  29  42 125 112      98  54  82   1   3       2  23  46  83  99     117  14  51  78  93
 85  67 114  84  52      15  66  55  79  80      33  57 100  71  59      77  65  73 116  10      90 110  35  68 124
 61  45  40   8  39      21  20 119  16  70     108 101  31  92  69      74  89  58  96  49      62   5  75   7 115
 76  28   6   9  94     123  41  50  56  26     103   4 113  22  25      91  72  12  95  13     118  36  18 106  63
 27 102 107 122  47      97 104  37  19  11      30  87  32 121  64      48  17  60  34  43      44  81 105  88 111
================================================================================================================
End state 1:
 16  91 106  92  10      49  51  45   1  15     109  40  96 113  52      60  13  37  97 119      81 120  47  12  55
 95  54  42 105  58      83   3  43  21 103      38  29  85  61 102      20  72 121  25  98      79  46  24  27 107
111  28 100  19  57      76  70  94 117  93      31  17  35  74  67       9  48  11  33  32      88  23  59  14  66
 89  18  36 104  68     101  63 125  86  78      84 114   2   7 108     110  44  69 122  71      87  99 123  65  39
 82 124  56 118  41       6   4   8  90  26      53 115 112   5  30     116  73  77  22  50      75  34  62  80  64
================================================================================================================
End state 2:
  4   3  94 115 101     123  57  10  90  35      15  18  96 121  65     107  63   1  89  56     102 108 114 122  69
 88 120 112  44  75      82  85  16  71  61      48  40  42  27   8       2  59  74  80 100      33  70  25  93 111
 98 117  11  46  43       5  39  12 106  51      52 105 109  21  20      36 119  23  87  50     124  81  26  55  29
 95  22  14  24  34      73  68  31  79  64     103  17  49  47 118      45 116  13  72  92      37  28  67  76   7
 30  53  84  86  62      77  66  38   9 113      97  58  19  78 104     125  60  91 110  54      99   6  83  32  41
================================================================================================================
End state 3:
101  75  19  20 100      10  67  47   3  31     103 122 124  84  58      13  23  56 113 110      88  28  69  95  16
118   8  17  91  81      11  96 105 120  85       7  21 111   5  73      92 115   2  42  82      87 116  80  57  26
 54   1 106  43  49      64  15  78  40  37      98  33  27  29  68      46  41  60  61 125      53  14  44  86  36
 24 114 108 123  66      30  22  76  59  72      35  89 112 102  70     107  32  97  94  50     119  39  34  55  63
 18 117  65  38  77      99   4   9  93  90       6  45  48  25  62      71 104 109  83  74     121  79  51  52  12
================================================================================================================
```

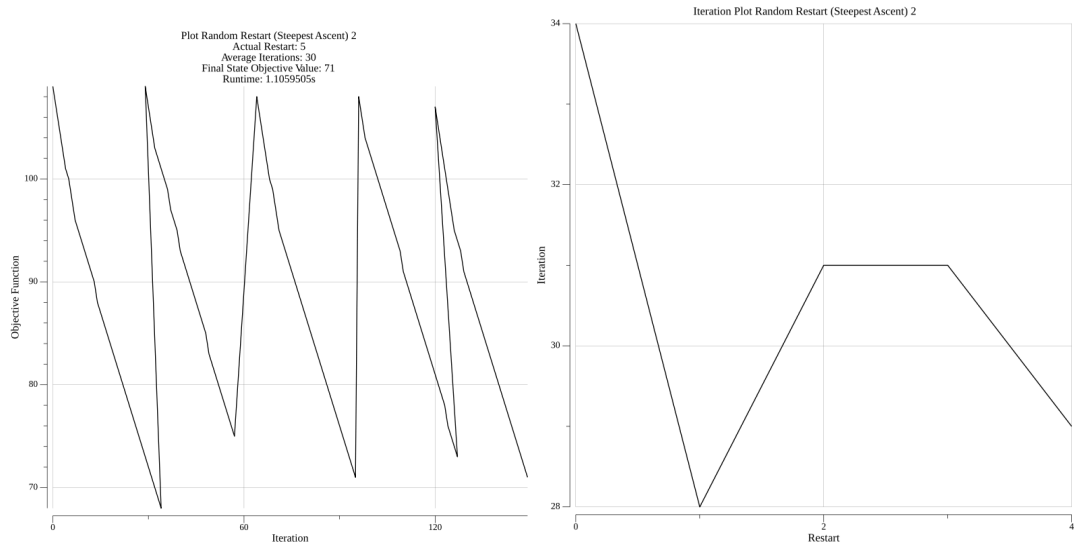
ALGORITHM	RUN	TIME	VALUE	V1	V2	ITERATION	STUCK	POPULATION
Initial			109	7061	718983			
RR (sta)	1	0.76	64	4757	486431	3*35 (avg)		
RR (sta)	2	1.11	71	5283	591169	5*30 (avg)		
RR (sta)	3	1.57	73	5485	608203	7*31 (avg)		

Percobaan 1 (Max Restart = 3)

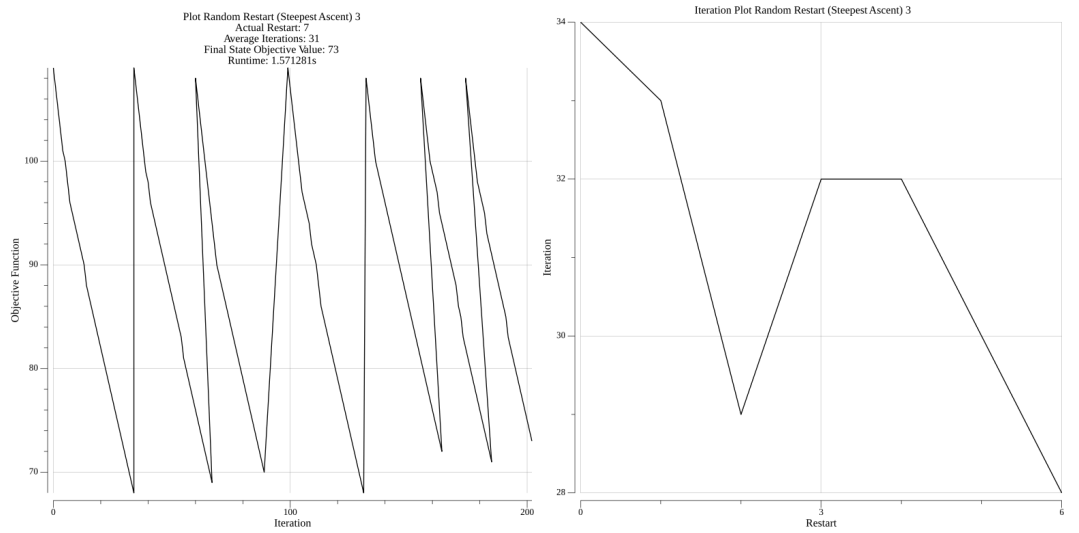




Percobaan 2 (Max Restart = 5)



Percobaan 3 (Max Restart = 7)



b. Analisis Hasil

Algoritma Random Restart Hill-Climbing dengan metode Steepest Ascent digunakan untuk mengatasi kemungkinan terjebak di *local optima*. Steepest Ascent adalah algoritma yang mencari solusi terbaik dengan melihat nilai *neighbor* yang memiliki value yang lebih tinggi secara iteratif. Namun, Steepest Ascent dapat terjebak dalam *local optima* terutama dengan ruang pencarian yang sangat besar. Di sisi lain, Random Restart dengan metode Steepest Ascent akan digunakan untuk

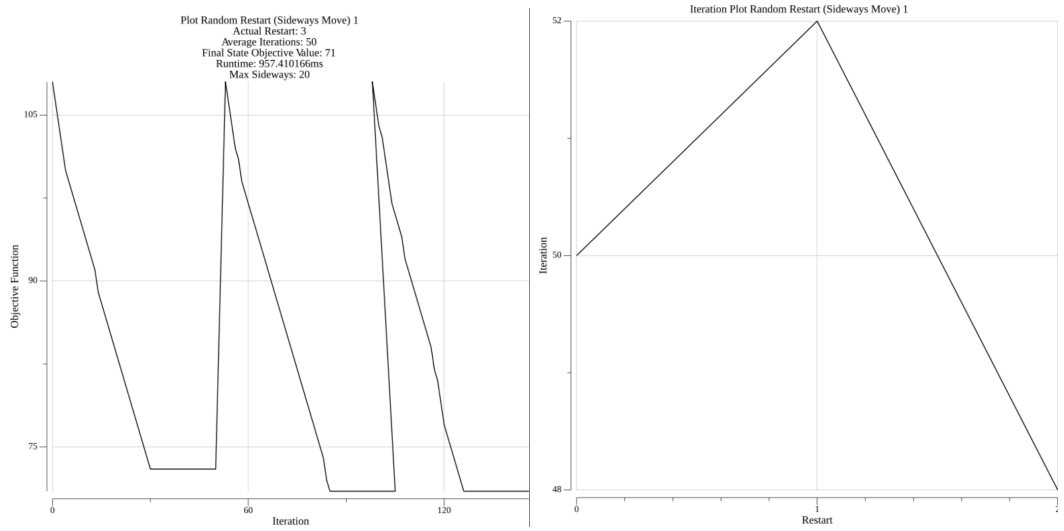
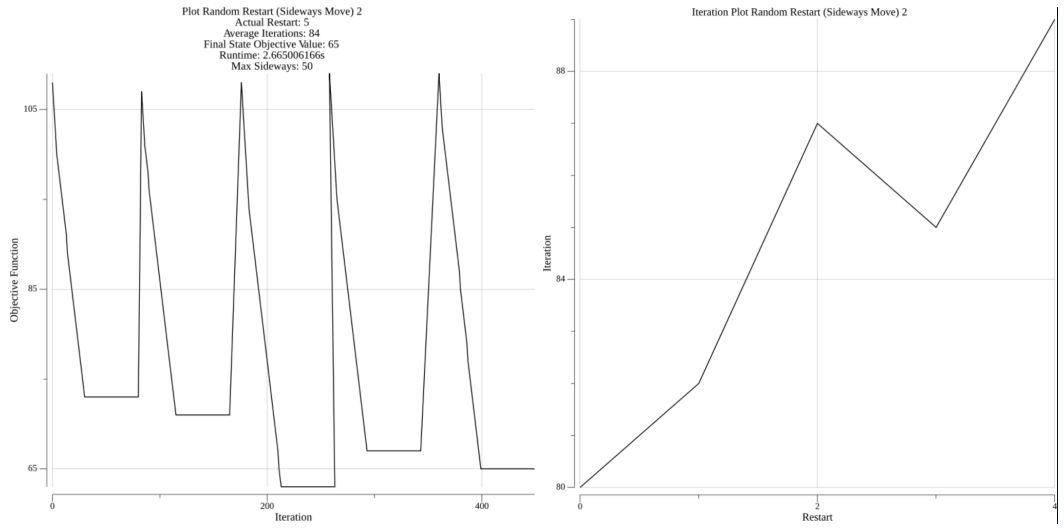
mengatasi masalah tersebut dengan melakukan *restart* beberapa kali dengan posisi acak. Dengan demikian, metode Random Restart dapat memberikan hasil yang mendekati *global optima* dengan memulai pencarian dari berbagai titik acak meskipun memiliki kemungkinan terjebak di *local optima*.

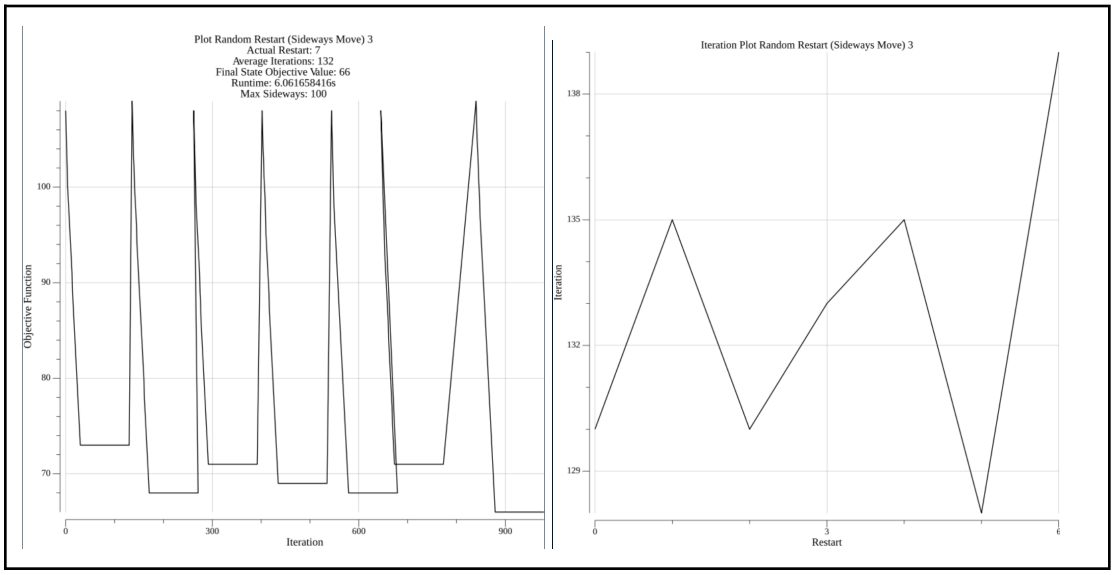
Berdasarkan *value* yang dihasilkan, dapat dilihat Random Restart dengan metode Steepest dapat menghasilkan nilai yang lebih optimal yaitu 64 dibandingkan dengan Steepest Ascent yang hanya dapat menghasilkan nilai 67. Meskipun demikian, nilai yang didapatkan terdiri dari 64, 71, dan 73 dari 3 percobaan yang telah dilakukan. Hal ini menunjukkan nilai yang dihasilkan oleh Random Restart belum memberikan hasil yang cukup dekat dengan *global optima*. Waktu yang dibutuhkan untuk menjalankan setiap Random Restart berkisar dari rentang 0.76s hingga 1.57s.

4. Random Restart Hill-climbing (Sideways)

a. Hasil Eksperimen

```
==================================RANDOM RESTART (SIDEWAYS MOVE)==================================
Enter max RESTART for rr_sm1: 3
Enter max RESTART for rr_sm2: 5
Enter max RESTART for rr_sm3: 7
Enter max SIDEWAY MOVES for rr_sm1: 20
Enter max SIDEWAY MOVES for rr_sm2: 50
Enter max SIDEWAY MOVES for rr_sm3: 100
Running random restart (sideways move)...
Running iteration 1 of random restart (sideways move)...
Running iteration 2 of random restart (sideways move)...
Running iteration 3 of random restart (sideways move)...
Initial state:
 73 111  19  81   5     100  82  13 105 124       6  70  55  67 112      90  93  21  87 102      18  71  24  57  15
 48  36  78  32  94      39  76 110  34  58      89  69 109  98  92     101  38  12   9   7     104   2  97  11  75
 37 119  83  22  74      96  44  52 120 122      60  23  88  25  72      42   4  80  86 107      16  77 108  49  68
 10  99 113  30  50      20 106  85  51 103      29  84  43 116  41      65  61  47  27  54      17  46 118  14   3
 35 125  26  59  53     121  31  40  33  62     123  66  45  28  64      63   1  56  91  79     115   8 117 114  95
==========================================================================================================
End state 1:
113  25  69  27  81      95  34  76  19  79     120  44  39  72  40     106 112 110  23  99     111 100   3  85  16
 50  64  33 125  87      75  74  66  14  86      38   1 114  62  36     109  57  18 103  10      43 119  88  11  54
 12  63   5  17  37      45 105  49 107  42      48  73   8 102  60       2 122  71  82 115      97 121  29   7  61
 89  70  92  78  56      46 104  28  84  53      77  30  52  32 124      94  20  26 108  67       9  35 117  13  15
 51  93 116  68  55      58  98  96  91  80     101   6 118  59  31      22   4  90  65  24      83 123  41  47  21
==========================================================================================================
End state 2:
 11  85 118  48  53      23  12  55  91  31      73  66  68  75  33     116  69  93  86  41      92  83 114  15  26
 27  77  30  78 103     117  56   3  72  67      43  28 121  94  29      47  49  62  80  82      81  79  99  16  40
 96 120  70  19  10       1  46  95  58 115     123 107  51  17 105      24   6  38  63  45      71  36  61  18  59
102  50   8  57  98      52  97  74  90   2      32  39  14 119 111       4  21 109  42  35     125 108 110   7  60
 64 122  89 113 100      20  54  88 106   5      44  65  84  34  22     124  87  13  37 112     104   9 101  25  76
==========================================================================================================
End state 3:
 80  58  75   6  96      24  59  37 105  90      98   5  64 103  45      42  15  22  69  41      71 112  19  32  43
 30   7   1  10  31      91  56  81  12  11      53  77 123  89  94      67  66   2  88  92      74 106 108 116  87
 82  86  60 117 102      62 120 115 125  55      63  33  49  83  38      72   3 118   4  95      36  73  79  76  25
 23 107  70  97  18      14  35  28  27  99      40 111  13  48 121      34  16 122 104   9      61 114  44  39  68
100  57 109  85 101     124  20  54  46  93      21 113 119   8  17      52  84  51  50  78      47  29  65 110  26
==========================================================================================================
==========================================================================================================
ALGORITHM               RUN     TIME    VALUE   V1      V2      ITERATION       STUCK   POPULATION
==========================================================================================================
Initial                                 108     6673    658075
RR (sm)                 1       0.96    71      5230    589520  3*50
RR (sm)                 2       2.67    65      4497    486407  5*84
RR (sm)                 3       6.06    66      4432    463530  7*132
```

Percobaan 1 (Max Restart = 3; Max Sideways = 20)

Percobaan 2 (Max Restart = 5; Max Sideways = 50)

Percobaan 3 (Max Restart = 7; Max Sideways = 100)



b. Analisis Hasil

Algoritma Random Restart Hill-climbing dengan metode Sideways Move digunakan dengan tujuan untuk mengatasi kekurangan algoritma Hill-climbing with Sideways Move yang memiliki kemungkinan terjebak di *local optima*. Algoritma ini akan melakukan *restart* beberapa kali di tempat yang acak dengan harapan dapat memberikan hasil yang lebih mendekati *global optima* meskipun tetap memiliki kemungkinan terjebak di *local optima*.

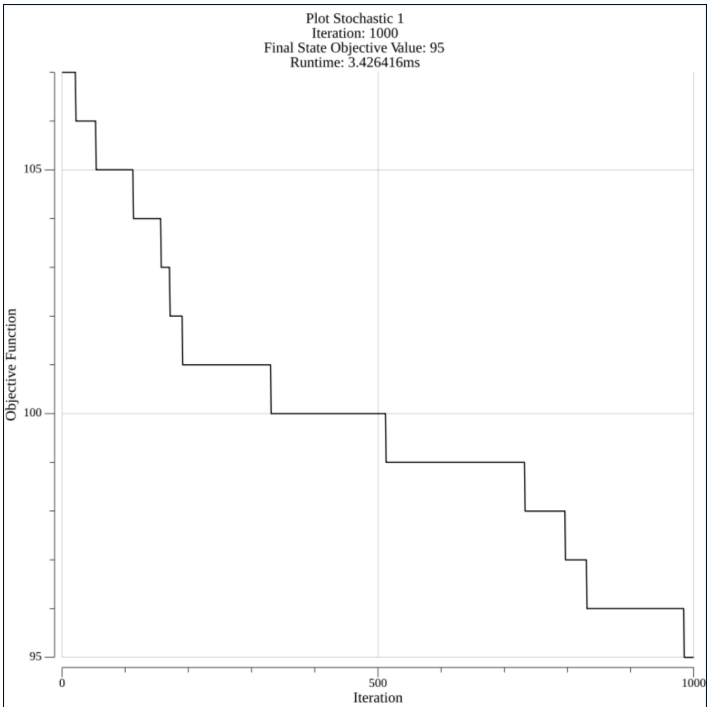
Berdasarkan hasil *value* akhir yang dihasilkan oleh algoritma Random Restart Hill-climbing dengan metode Sideways Move, dapat dilihat bahwa hasilnya adalah 71, 65, dan 66 yang secara rata-rata lebih baik daripada algoritma Hill-climbing with Sideways Move yang memiliki *value* akhir sebesar 70. Waktu yang dibutuhkan untuk menjalankan algoritma berkisar dari rentang 0.96s hingga 6.06s. Namun, nilai yang dihasilkan oleh algoritma ini belum memberikan hasil yang cukup dekat dengan *global optima*.

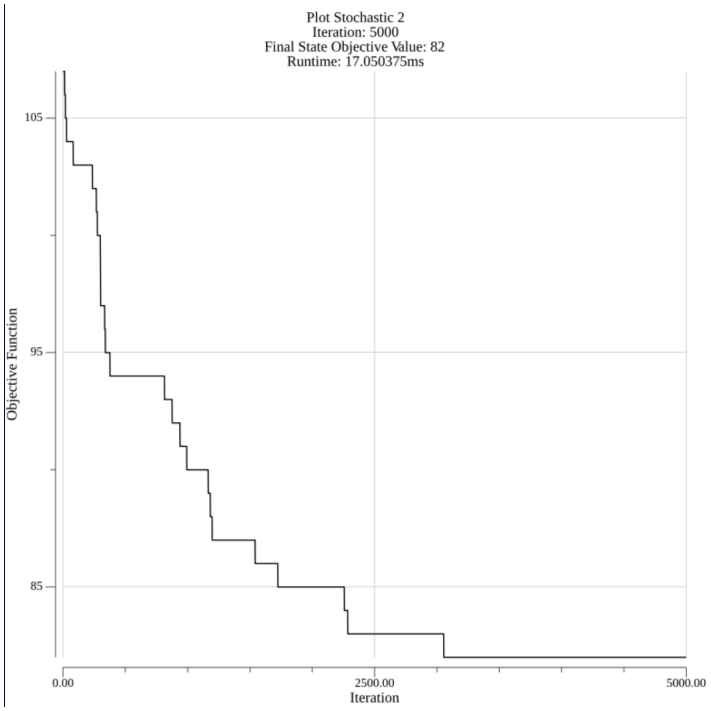
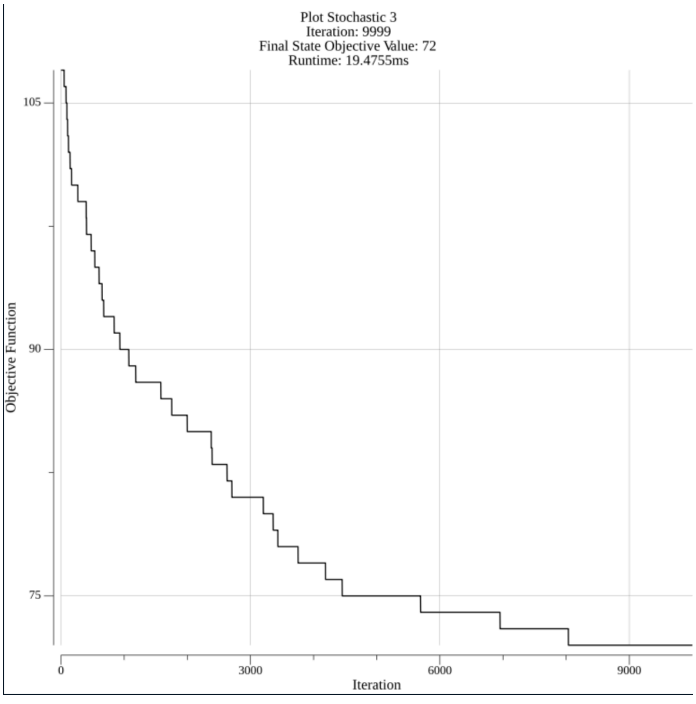
5. Stochastic Hill-climbing

a. Hasil Eksperimen

```
==================================STOCHASTIC==================================
Enter max iterations for s1: 1000
Enter max iterations for s2: 5000
Enter max iterations for s3: 9999
Running stochastic...
Initial state:
 79  15  49  82  98      35  11 105  44  36      25  59 119 120 112      62  71 125  22   7      60 114  76  13  54
110 107  45  23  67      29  10 124  47   6       8  91  94  68   2      40  64  99 113  92      21  69  78  12  87
 93  55  86  96  20     103  24  42  43   1      66  17 102 123  39     116  80 111  14  53      70  34  16  38  95
 83 108 117 115  19      89  28  41  56  77     118  88 109  58   5      50  74  37  48   9      84  72  51 101  61
 73  32  27 122 121      52  33 100  63   3      26 104  90  31  81      75  30 106  97  85      57   4  65  46  18
================================================================================================================
End state 1:
 79  15 123  82  98      35  11 105  44  68      25  59 119 120 112      62  71 125  50   7      60 114  76  13  54
110 107  45  23  67      29  10 124  47  99       8  91  94  36  78      40  64   6 113  92      21  69   2  12  87
 93 118  86  96  20     103  24  42  43   1      66  17 102  49  81     116  80 121  14  53      70  34  16 101  95
 37 108 117 115  19      89 106  41  56  77      55  88 109  58   5      22  74  83  48   9      84  72  51  38  61
 73  27  32 122 111      52  33 100  63   3      26 104  90  31  39      75  30  28  97  85      57   4  65  46  18
================================================================================================================
End state 2:
 79  15  40  82  98     108  24 103  36  44      25  59 119  30 112      62 124   1  39   7      83 114  46  13  54
110 107  45  23 106      63  68  71  91  78      97  47  94  10   2      12  64  99  48  92      21  69   6  49  87
 93  55  86  96  20     105  11  42  43 125      66  17 102 123  22     116  80 111  14  53      70  34  16  38  95
 29  35 117 115  19     113  28  41  56  77     118  88 109 121   5      50  74  37  89   4      84  72  51   8  61
 73  32  27 122  58      52  33 100  60   3       9 104  90  31  81      75 120  67 101  85      57  26  65  76  18
================================================================================================================
End state 3:
 79   7  49  82  98      89  69   2  44 111      25  88 119 120  51      62  71 125  56   1      60 112  76  13  54
110 107  14  27  57      29 109 124  47   6     115  73  94  68  85      40  64   5 113  92      21  12  78 117  87
 93  55  81  66  20     103  67  42  43  10      34  17 102 123  39      15  80  36  45  11      70  96  16  38  95
 83 108  97   8  19      35  28  41  22   9     118  33 116  58  99      50  74  37  48 106      84  72 114 101  61
 23  32  91 122 121      59  90 100  63   3      26 104  77  31  86      75  30  52  53 105      24   4  65  46  18
================================================================================================================
================================================================================================================
ALGORITHM               RUN     TIME    VALUE   V1      V2      ITERATION       STUCK   POPULATION
================================================================================================================
Initial                                 107     7371    766427
Stochastic              1       0.00    95      6564    694906  1000
Stochastic              2       0.02    82      5468    533104  5000
Stochastic              3       0.02    72      4587    451143  9999
```

Percobaan 1 (Max Iteration = 1000)



Percobaan 2 (Max Iteration = 5000)

Percobaan 3 (Max Iteration = 9999)


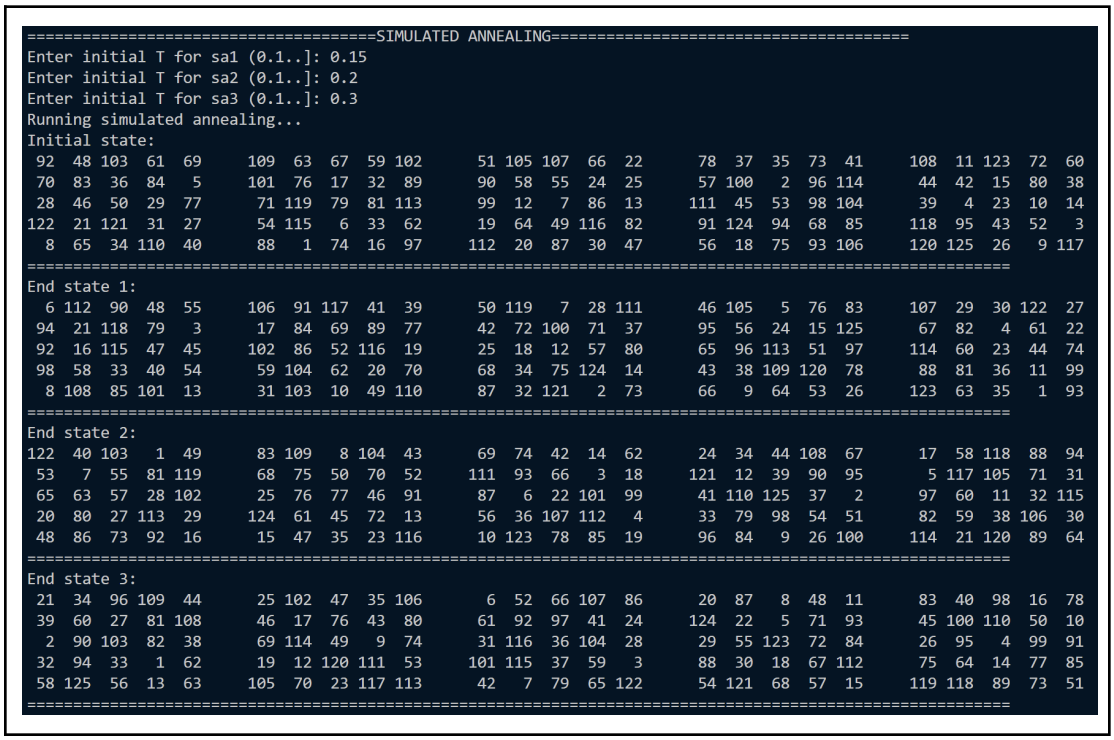
b. Analisis Hasil

Algoritma Stochastic Hill-climbing adalah algoritma pengoptimalan solusi yang menghasilkan solusi *neighbor* acak dari solusi *current* secara iteratif. Algoritma ini tidak selalu memilih solusi yang paling baik, tetapi dengan probabilitas tertentu yang memungkinkan algoritma untuk menjelajahi lebih luas menghasilkan solusi yang memuaskan dan menghindari puncak lokal.

Berdasarkan hasil *value* akhir yang dihasilkan oleh algoritma Stochastic Hill-climbing, dapat dilihat bahwa hasilnya adalah 95, 82, dan 72 dengan maksimum iterasi sebesar 1000, 5000, dan 9999. Waktu yang dibutuhkan untuk menjalankan setiap percobaan berkisar dari rentang 0s hingga 0.02s yang mana bisa dikatakan sangat cepat. *Objective value* yang dihasilkan oleh algoritma ini memang semakin baik seiring bertambahnya jumlah maksimum iterasi. Namun, saat mencapai angka iterasi yang tinggi, algoritma Stochastic Hill-climbing mulai terjebak di *local optima* sehingga bisa disimpulkan bahwa algoritma ini belum bisa memberikan hasil yang cukup dekat dengan *global optima*.

6. Simulated Annealing

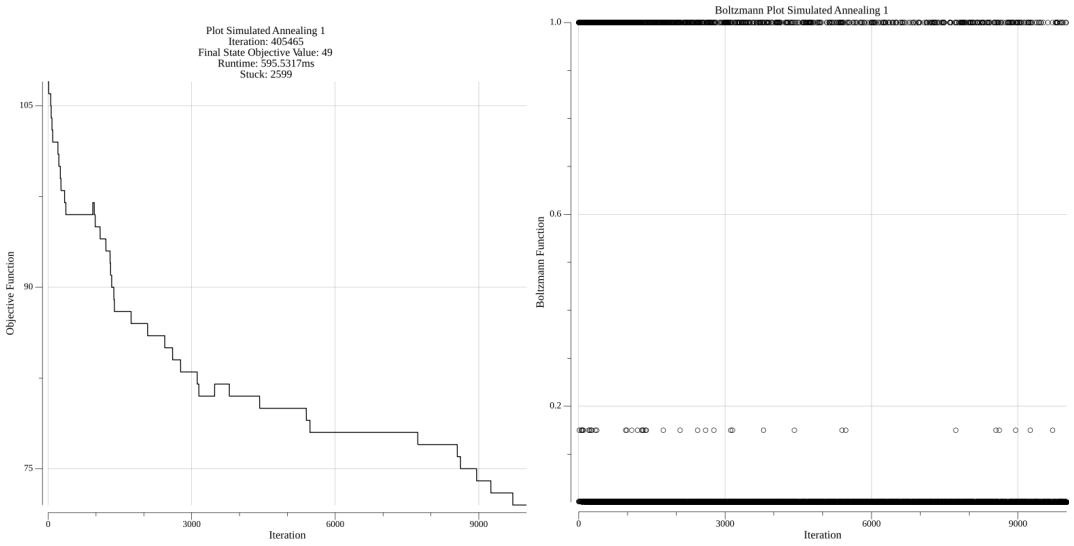
a. Hasil Eksperimen



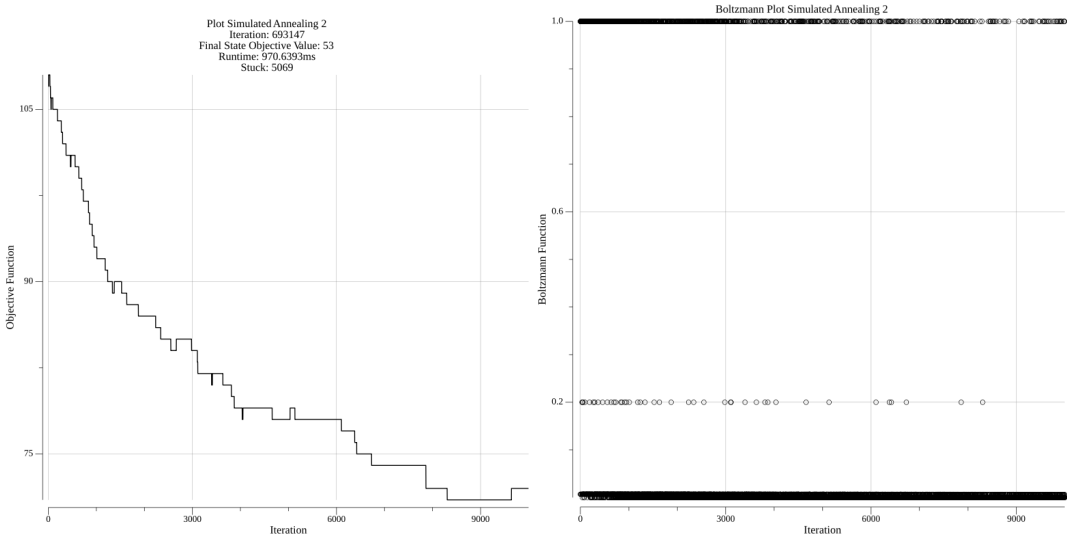
```
====================================SIMULATED ANNEALING=====================================
Enter initial T for sa1 (0.1..]: 0.15
Enter initial T for sa2 (0.1..]: 0.2
Enter initial T for sa3 (0.1..]: 0.3
Running simulated annealing...
Initial state:
 92  48 103  61  69     109  63  67  59 102      51 105 107  66  22      78  37  35  73  41     108  11 123  72  60
 70  83  36  84   5     101  76  17  32  89      90  58  55  24  25      57 100   2  96 114      44  42  15  80  38
 28  46  50  29  77      71 119  79  81 113      99  12   7  86  13     111  45  53  98 104      39   4  23  10  14
122  21 121  31  27      54 115   6  33  62      19  64  49 116  82      91 124  94  68  85     118  95  43  52   3
  8  65  34 110  40      88   1  74  16  97     112  20  87  30  47      56  18  75  93 106     120 125  26   9 117
================================================================================================
End state 1:
  6 112  90  48  55     106  91 117  41  39      50 119   7  28 111      46 105   5  76  83     107  29  30 122  27
 94  21 118  79   3      17  84  69  89  77      42  72 100  71  37      95  56  24  15 125      67  82   4  61  22
 92  16 115  47  45     102  86  52 116  19      25  18  12  57  80      65  96 113  51  97     114  60  23  44  74
 98  58  33  40  54      59 104  62  20  70      68  34  75 124  14      43  38 109 120  78      88  81  36  11  99
  8 108  85 101  13      31 103  10  49 110      87  32 121   2  73      66   9  64  53  26     123  63  35   1  93
================================================================================================
End state 2:
122  40 103   1  49      83 109   8 104  43      69  74  42  14  62      24  34  44 108  67      17  58 118  88  94
 53   7  55  81 119      68  75  50  70  52     111  93  66   3  18     121  12  39  90  95       5 117 105  71  31
 65  63  57  28 102      25  76  77  46  91      87   6  22 101  99      41 110 125  37   2      97  60  11  32 115
 20  80  27 113  29     124  61  45  72  13      56  36 107 112   4      33  79  98  54  51      82  59  38 106  30
 48  86  73  92  16      15  47  35  23 116     10 123  78  85  19      96  84   9  26 100     114  21 120  89  64
================================================================================================
End state 3:
 21  34  96 109  44      25 102  47  35 106       6  52  66 107  86      20  87   8  48  11      83  40  98  16  78
 39  60  27  81 108      46  17  76  43  80      61  92  97  41  24     124  22   5  71  93      45 100 110  50  10
  2  90 103  82  38      69 114  49   9  74      31 116  36 104  28      29  55 123  72  84      26  95   4  99  91
 32  94  33   1  62      19  12 120 111  53     101 115  37  59   3      88  30  18  67 112      75  64  14  77  85
 58 125  56  13  63     105  70  23 117 113      42   7  79  65 122      54 121  68  57  15     119 118  89  73  51
================================================================================================
```

ALGORITHM	RUN	TIME	VALUE	V1	V2	ITERATION	STUCK	POPULATION
Initial			107	6824	637088			
Simulated annealing	1	0.60	49	3588	390566	405465	2599	
Simulated annealing	2	0.97	53	3468	345212	693147	5069	
Simulated annealing	3	1.72	46	3494	372292	1098612	67104	

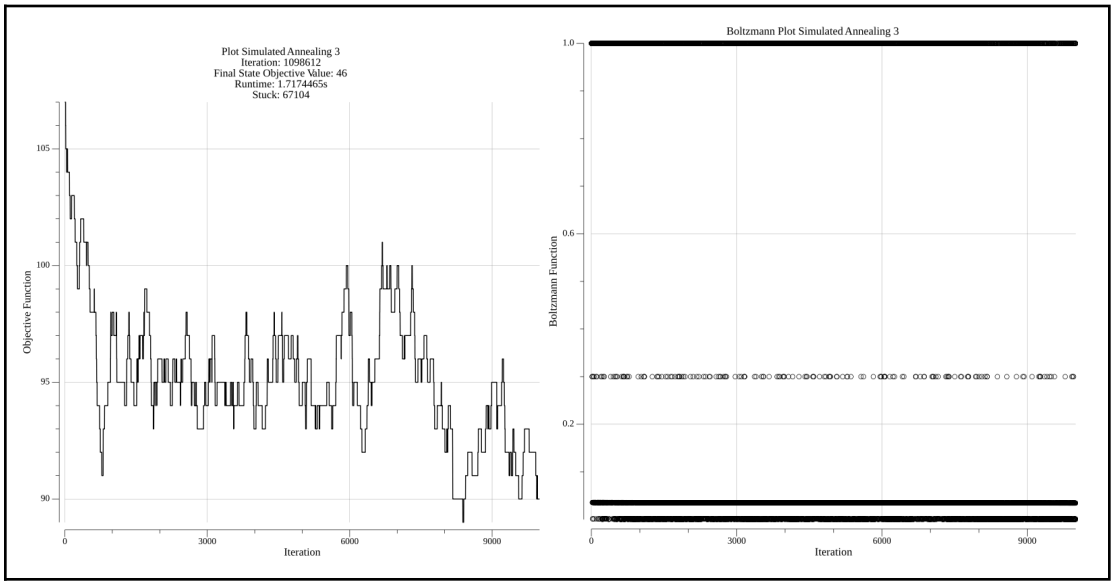
Percobaan 1 (InitialT = 0.15)



Percobaan 2 (InitialT = 0.2)



Percobaan 3 (InitialT = 0.3)



b. Analisis Hasil

Dari hasil percobaan, algoritma Simulated Annealing menunjukkan performa yang baik dalam mencari solusi dengan meminimalkan *objective function* (107 menjadi 46 pada percobaan ketiga). Variasi InitialT berpengaruh dalam proses pencarian dan hasil yang diperoleh. Suhu awal yang tinggi memungkinan algoritma untuk mencari solusi lebih luas sehingga solusi yang lebih baik dapat dicapai meskipun membutuhkan waktu yang lebih lama. Simulated Annealing juga dapat menghindari jebakan pada *local optima* dengan mengizinkan solusi lebih buruk untuk sementara waktu sehingga dapat menemukan solusi yang *global optima*.

Secara keseluruhan, Simulated Annealing merupakan metode optimasi yang efektif dalam mengatasi masalah dengan ruang yang kompleks seperti adanya penurunan nilai secara signifikan setelah beberapa iterasi. Value yang didapatkan dari 3 percobaan secara berurutan yaitu 49, 53, dan 46. Waktu yang dibutuhkan untuk menjalankan algoritma ini cukup cepat berkisar dari 0.6s hingga 1.72s.

7. Genetic Algorithm

a. Hasil Eksperimen

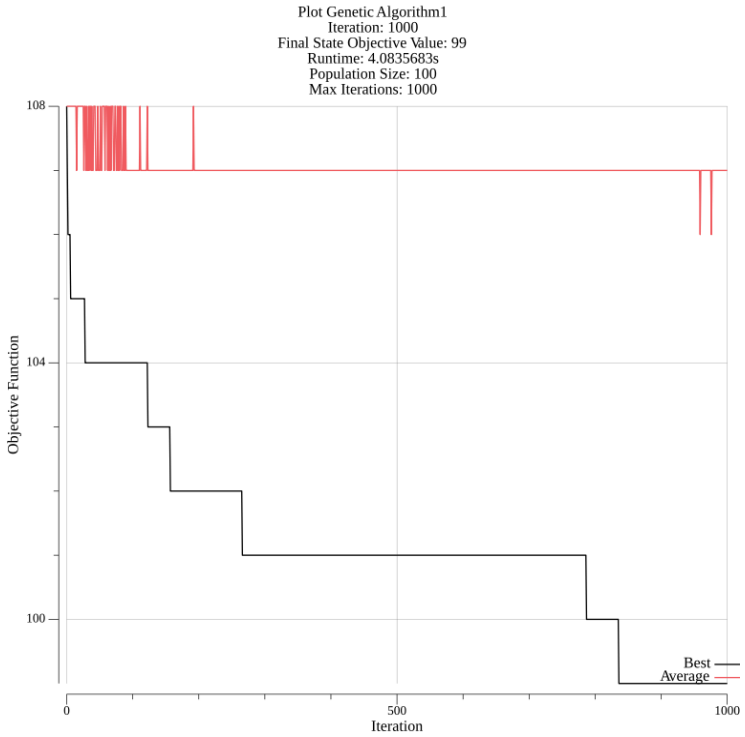
Percobaan dengan parameter jumlah populasi = 100

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

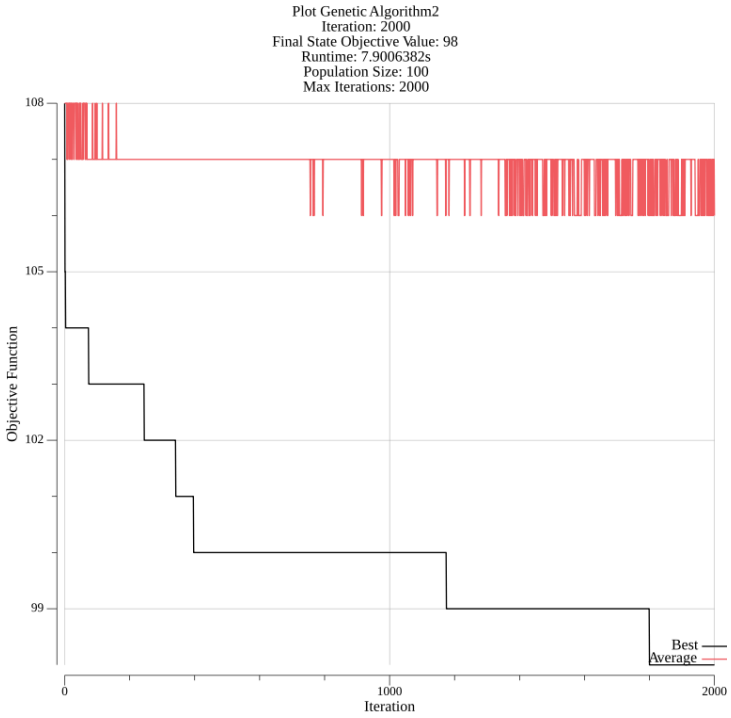
```
=============================================================================================================
ALGORITHM               RUN      TIME    VALUE   V1      V2      ITERATION       STUCK   POPULATION
=============================================================================================================
Initial                                  108     6640    591588
Genetic Algorithm       1        4.08    99      6259    636147  1000                    100
Genetic Algorithm       2        7.90    98      6290    599556  2000                    100
Genetic Algorithm       3        12.83   96      6920    720214  3000                    100
=============================================================================================================
```

Grafik percobaan dengan parameter jumlah populasi = 100

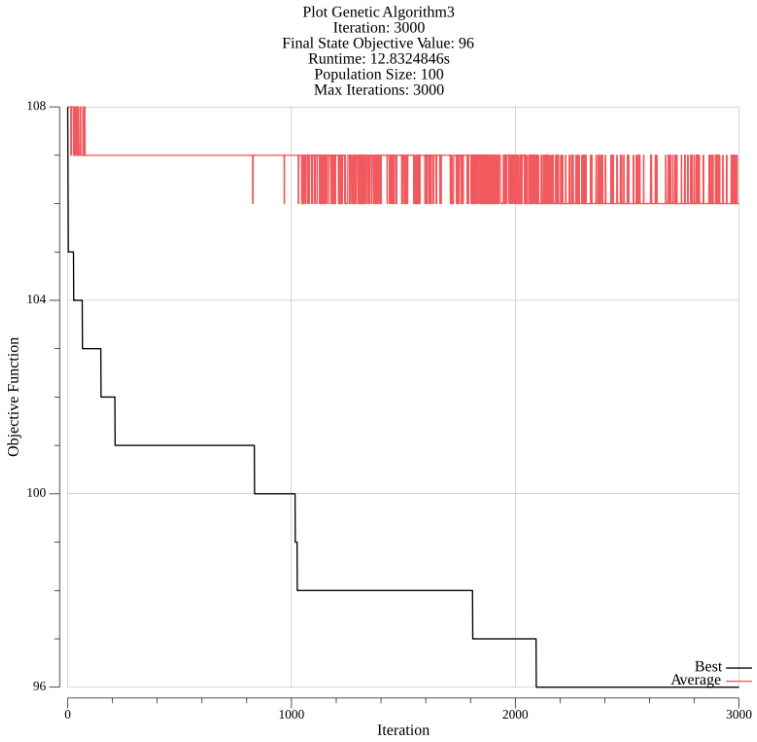
Percobaan 1 : Max Iteration = 1000



Percobaan 2 : Max Iteration = 2000



Percobaan 3 : Max Iteration = 3000



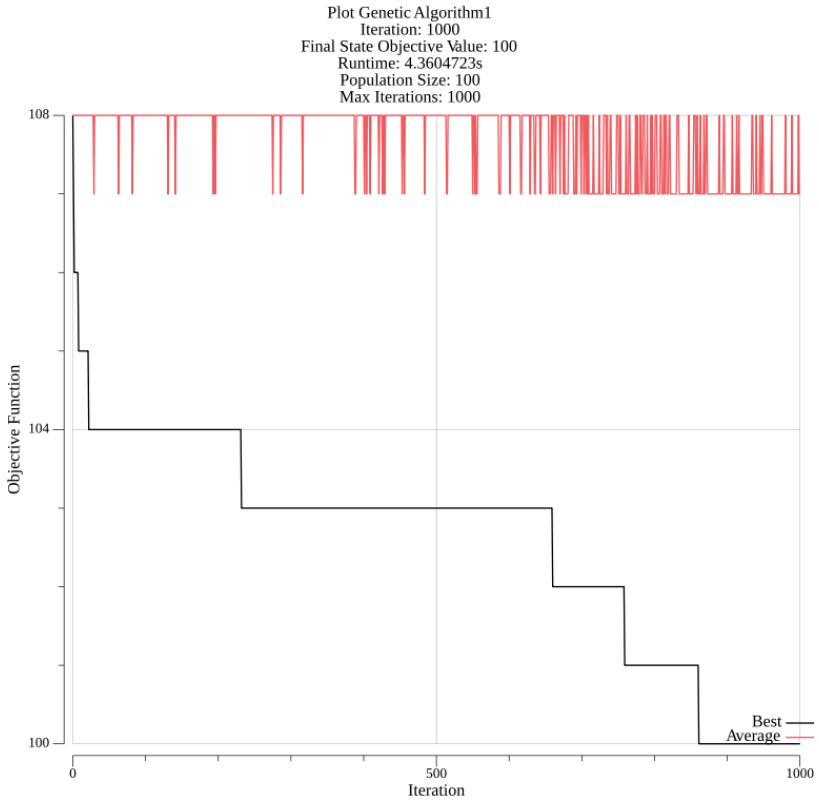
Percobaan dengan parameter jumlah Max Iteration = 1000

```
Enter POPULATION SIZE for ga1: 100
Enter POPULATION SIZE for ga2: 200
Enter POPULATION SIZE for ga3: 300
Enter MAX ITERATIONS for ga1: 1000
Enter MAX ITERATIONS for ga2: 1000
Enter MAX ITERATIONS for ga3: 1000
Running genetic algorithm...
Generating dump file...
Generating plot file...
Initial state:
122  35  29  24 117      20  82  23  40 111      69  94   8  26 114     113  97  60  17  93      21  99 105   4  27
  6  50  45  71  48      65  11 112  72  95      74  61  84  98  10       3 102 119  37  41      96  32 101  73  36
116  54  76 125  47     123  18  34 100 108       9  91  31 110  81      86  63   2  68  55      49 109 103  46  51
 33  85 115 120  70      25  66   1  39  75      57  38   5  62  92     121  67  16  58  56      89  13  64  83  14
 22  78  90 107 118      43  88  12  42  30      59 124 104  44  28      77  19  53  80 106      87  79   7  52  15
================================================================================================================
End state 1:
 93  28 125   4 106      32  53  47  23 105      20  86 101  79 120      31  50  73  78  14      35  82  37  54 107
103  27  10  44  55     102  91  70  11  52       2 100 110  76  34      24 104  66  25   7      19  71 124  92 108
  1  38  59  77  17      57  84  67   8  99      74  64  89  75  13     121  63  49 115 117       9  69  83  87  16
  6   5  85  22  33      26  62 114  21  60     109  12  68  15  36      18  58  30 122  43     111  65  46 123 113
 45   3  90  98 119      61  72  88  39  40     116  81  29  80 112      42  95  97  41 118      51  96  48  94  56
================================================================================================================
End state 2:
 61  13 101  36  60      56  71   5  57  82      43  12  96   2  26     107  97  63  53  78      84 123  17  94  73
 19 114  32  27  69      16  51  72   7  31     124  20  75  54  42      30  77  37  48 117      93  28 116  40  38
 58  47  92 103  62      29 121  80  98  65       8  99 100  74   9     118  22  46   4 105      68  59  76  89  83
 55  95 104  52  87      70 102  23 113 108      91  90  33  67  50      21  64  49  85 110     115  81  79   3   1
122   6  35 111  41      14  10 119  66  15      34  45  44  86 106      24 112  18 125  11     109  88  39  25 120
================================================================================================================
End state 3:
 76  10   4  89 125      75  19  71  25  61      85  11  95   3  34     122  26  68  50  80      84  29  54 108  51
  7  62 114  59  48     102  82 120 116  24      57  49  93 110  37      53  52  28  67  70     100  72  39  92  66
 33  43  27 109 103     111 105  69  32  30      16 106  17  31 118      73 115  13 104 123      45  98 101  56  15
 44   1  58   6   2      97   9  79  38  87     117  23 121  78 119      88 107  90  86  21      47  91  74  55  35
 65  36 112  99  46      60  94  81   8 113      40  41  18  14  77      64  63  96  12  42      22 124  20   5  83
```

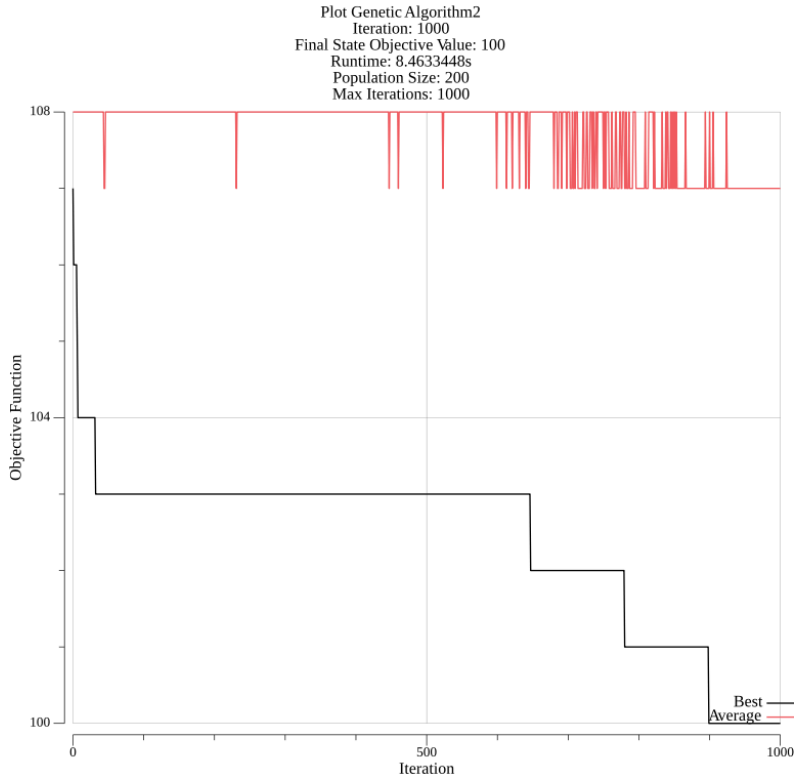
ALGORITHM	RUN	TIME	VALUE	V1	V2	ITERATION	STUCK	POPULATION
Initial			109	6782	666886			
Genetic Algorithm	1	4.36	100	6326	597666	1000		100
Genetic Algorithm	2	8.46	100	5291	423911	1000		200
Genetic Algorithm	3	13.18	99	6112	591980	1000		300

Grafik percobaan dengan parameter Max Iteration = 1000

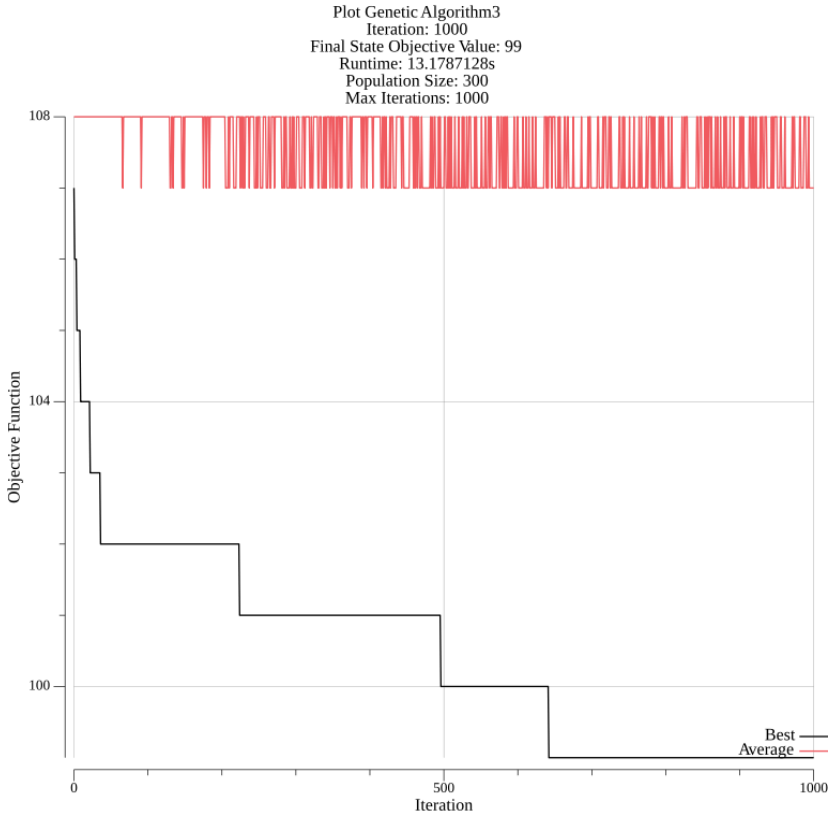
Percobaan 1 : Population = 100



Percobaan 2 : Population = 200



Percobaan 3 : Population = 300



b. Analisis Hasil

Berdasarkan hasil percobaan, bisa dilihat bahwa value terbaik yang bisa dihasilkan oleh Genetic Algorithm adalah 96 di mana angka tersebut sangat jauh dari *global optima*. Pada algoritma ini, semakin tinggi jumlah maksimum iterasi, semakin baik pula hasil yang diberikan. Namun, dengan meningkatnya jumlah maksimum iterasi, waktu yang dibutuhkan untuk menyelesaikan proses pencarian akan semakin lama. Selain itu, Genetic Algorithm seringkali terjebak di *local optima* karena ruang lingkup pencarian yang sangat besar yang bisa dilihat dari banyaknya gerakan mendatar pada grafik.

KESIMPULAN DAN SARAN

I. Kesimpulan

- Algoritma *local search* cenderung sulit untuk mencapai kondisi *global optima* apabila terdapat pada masalah yang memiliki banyak puncak lokal, seperti Diagonal Magic Cube. Hal ini terjadi pada algoritma Steepest Ascent Hill-climbing dan Hill Climbing with Sideways Move yang tidak memiliki teknis yang cukup untuk dapat keluar dari local optima nya.
- Dalam persoalan waktu pencarian solusi diagonal *magic cube* dengan *local search*, algoritma yang memiliki waktu pencarian solusi tercepat adalah Stochastic Hill-climbing dalam waktu 0.0s dan Genetic Algorithm adalah algoritma yang membutuhkan waktu paling lama untuk menghasilkan solusi yaitu 13.18s.
- Dari segi program penyelesaian permasalahan Diagonal Magic Cube 5 x 5 x 5 ini, dapat disimpulkan bahwa hasil dari algoritma Simulated Annealing memberikan performa paling baik dengan solusi yang mendekati global optima. Algoritma lain seperti Steepest Ascent Hill-climbing, Hill Climbing with Sideways Move, Random restart Hill-climbing, dan Genetic Algorithm menunjukkan hasil pencarian yang kurang optimal dilihat dari hasil akhir yang sering terjebak di local optima dengan nilai akhir yang lebih buruk.

II. Saran

- Melakukan efisiensi pada kode program untuk mendapatkan hasil (*value*) yang lebih baik. Misalnya, dengan menyimpan serangkaian angka yang telah mencapai 315 pada algoritma tertentu. Lebih spesifik lagi, algoritma *genetic algorithm* dapat memanfaatkan elitism dan mutation rate dinamis yang beradaptasi terhadap konvergensi hasil *cube*. Saat ini, sudah terdapat mekanisme sederhana mutation rate dan elitism dinamis. Namun, eksperimen khusus yang lebih berfokus untuk menemukan fungsi khusus mutation rate dan elitism mungkin akan meningkatkan hasil *genetic algorithm*.
- Menggunakan algoritma atau pendekatan lain untuk pencarian solusi magic cube 5 x 5 x 5.

PEMBAGIAN TUGAS

NIM	Nama	Tugas
18222034	Christoper Daniel	Random Restart Hill-climbing (Sideways), Simulated Annealing
18222035	Lydia Gracia	Stochastic Hill-climbing, Genetic Algorithm, Video Player
18222049	Willhelmina Rachel Silalahi	Steepest Ascent Hill-climbing, Hill-climbing with Sideways Move
18222100	Ervina Limka	Random Restart Hill-climbing (Steepest), Simulated Annealing

REFERENSI

[1] Ambalavanan, V., & Chitra, P. (2018). A performance analysis of simulated annealing and genetic algorithms for the optimization of NP-hard problems. *International Journal of Education and Management Engineering (IJEME)*, *8*(1), 1-10. https://doi.org/10.5815/ijeme.2018.01.01

[2] Chou, J. S., & Ngo, N. T. (2019). Adapting mutation rates for the dynamic optimization of traveling salesman problems. *Information*, *10*(12), 390. https://doi.org/10.3390/info10120390

[3] Cotta, C., & Troya, J. M. (2019). Fast convergence in genetic algorithms with elitism: Analysis and application. *Symmetry*, *11*(9), 1145. https://doi.org/10.3390/sym11091145

[4] Deb, K., & Agrawal, S. (1995). Elitism for convergence of evolutionary multi-objective optimization. *Polish Energy*, *2012*(1b), 60. Retrieved from http://pe.org.pl/articles/2012/1b/60.pdf

[5] Fathoni, A., & Putra, B. R. (2019). Elitism parameter study for genetic algorithm optimization. *Journal of Informatics and Computer Science (JICON)*, *8*(2), 123-134. https://doi.org/10.35508/jicon.v8i2.2894

[6] Wikipedia contributors. (2023). *Crossover (genetic algorithm)*. In *Wikipedia, The Free Encyclopedia*. Retrieved from https://en.wikipedia.org/wiki/Crossover_(genetic_algorithm)#Order_crossover_(OX1)

[7] Wikipedia contributors. (2023). *Mutation (genetic algorithm)*. In *Wikipedia, The Free Encyclopedia*. Retrieved from https://en.wikipedia.org/wiki/Mutation_(genetic_algorithm)#Inversion