Anexo I – Verificar quantidade de CPUs disponíveis

```
package main

import (
      "fmt"
      "runtime"
)

func main() {

      runtime.GOMAXPROCS(runtime.NumCPU())

      fmt.Println("O numero de processadores disponíveis foi: ",
runtime.NumCPU())

}
```

Anexo II – Programa TESTE Sequencial

```
package main

import (
	"bufio"
	"fmt"
	"math"
	"math/rand"
	"os"
	"sort"
	"strconv"
	"strings"
	"time"

	"github.com/pmylund/sortutil"
)

type city struct {
	id                  int
	latitude, longitude float64
}

type chromosome struct {
	id      int
	fitness float64
	cities  []city
}

type set map[interface{}]bool

func main() {

	if len(os.Args) > 4 {
		fileDirectory, populationSize, generations, mutation :=
readArgs()
		searchInstance(fileDirectory, populationSize, generations,
mutation)

	} else {
		fmt.Println("Passe os argumentos para executar o experimento")
		fmt.Println("ARGS: fileDirectory, populationSize, generations,
mutation")
	}
}

func searchInstance(fileDirectory, populationSizeString,
generationsString, mutationString string) {
	fileCities := readCity(fileDirectory)
	cities := getArrayOfCities(fileCities)

	populationSize, err := strconv.Atoi(populationSizeString)
	if err != nil {
		fmt.Println("O tamanho da polulação é inválido\n", err)
		os.Exit(1)
	}
```

```
	generations, err := strconv.Atoi(generationsString)
	if err != nil {
		fmt.Println("O valor das gerações é inválido\n", err)
		os.Exit(1)
	}
	mutation, err := strconv.ParseFloat(mutationString, 64)
	if err != nil {
		fmt.Println("O valor das mutações é inválido\n", err)
		os.Exit(1)
	}
	var population []chromosome
	population = createInitialPopulationWithFitness(cities,
populationSize)

	for index := 0; index < generations; index++ {
		sortutil.AscByField(population, "fitness")

		population := elitism(populationSize, population)
		lenPopulationSelecionada := len(population)

		var percent = (populationSize) * 75 / 100
		if percent%2 != 0 {
			percent--
		}

		for len(population) < populationSize {

			var indexes = randomInts(2, 0, populationSize,
makeRandomNumberGenerator())
			sort.Ints(indexes)

			valor1 :=
ox(population[rand.Intn(lenPopulationSelecionada)].cities,
population[rand.Intn(lenPopulationSelecionada)].cities, indexes[0],
indexes[1])

			population = append(population,
mutate(createChromosome(valor1), populationSize, mutation))

		}
		fmt.Println(population[0].fitness)
	}

	fmt.Println(population[0].fitness)

}

func ox(p1, p2 []city, a, b int) []city {
	var (
		n  = len(p1)
		o1 = make([]city, n)
		o2 = make([]city, n)
	)

	copy(o1[a:b], p1[a:b])
	copy(o2[a:b], p2[a:b])
```

```
	var o1Lookup, o2Lookup = make(set), make(set)
	for i := a; i < b; i++ {
		o1Lookup[p1[i]] = true
		o2Lookup[p2[i]] = true
	}

	var j1, j2 = b, b
	for i := b; i < b+n; i++ {
		var k = i % n
		if !o1Lookup[p2[k]] {
			o1[j1%n] = p2[k]
			j1++
		}
		if !o2Lookup[p1[k]] {
			o2[j2%n] = p1[k]
			j2++
		}
	}
	return o1
}

func randomInts(k, min, max int, rng *rand.Rand) (ints []int) {
	ints = make([]int, k)
	for i := 0; i < k; i++ {
		ints[i] = i + min
	}
	for i := k; i < max-min; i++ {
		var j = rng.Intn(i + 1)
		if j < k {
			ints[j] = i + min
		}
	}
	return
}

func makeRandomNumberGenerator() *rand.Rand {
	return rand.New(rand.NewSource(time.Now().UnixNano()))
}

func mutate(gene chromosome, populationSize int, motation float64)
chromosome {
	if motation == 0 {
		return gene
	}

	for rand.Float64() < motation {
		position1 := rand.Intn(populationSize)
		position2 := rand.Intn(populationSize)

		aux := gene.cities[position1]
		gene.cities[position1] = gene.cities[position2]
		gene.cities[position2] = aux

		gene.fitness = calculateFitness(gene.cities)
	}
```

```
	return gene
}

func elitism(populationSize int, geracao []chromosome) []chromosome {
	var percent = (populationSize) * 25 / 100
	if percent%2 != 0 {
		percent++
	}
	return geracao[0:int(percent)]
}

func calculateTotalFitness(populationSize int, geracao []chromosome)
float64 {
	var totalFitness float64
	for _, valor := range geracao {
		totalFitness += valor.fitness
	}
	return totalFitness
}

func createInitialPopulationWithFitness(cities []city, populationSize
int) []chromosome {
	primeiraGeracao := []chromosome{}

	for index := 0; index < populationSize; index++ {
		primeiraGeracao = append(primeiraGeracao,
createChromosomeOfInitialPopulation(cities))
	}

	return primeiraGeracao
}

func createChromosomeOfInitialPopulation(cities []city) chromosome {
	tmp := make([]city, len(cities))
	copy(tmp, cities)

	individuo := shuffle(tmp)
	fitness := calculateFitness(shuffle(cities))
	return chromosome{fitness: fitness, cities: individuo}
}

func createChromosome(cities []city) chromosome {
	fitness := calculateFitness(cities)
	return chromosome{fitness: fitness, cities: cities}
}

func readArgs() (string, string, string, string) {
	return os.Args[1], os.Args[2], os.Args[3], os.Args[4]
}

func calculateFitness(cities []city) float64 {

	var length = len(cities) - 1
	var fitness float64
```

```
	for index := 0; index < length; index++ {
		fitness += calculateDistanceCoordenate(cities[index],
cities[index+1])
	}
	fitness += calculateDistanceCoordenate(cities[length], cities[0])
	return fitness

}

func calculateDistanceCoordenate(cidadeOrigem, cidadeDestino city)
float64 {

	return 6371 * math.Acos(math.Cos(math.Pi*(90-
cidadeDestino.latitude)/180)*math.Cos((90-
cidadeOrigem.latitude)*math.Pi/180)+math.Sin((90-
cidadeDestino.latitude)*math.Pi/180)*math.Sin((90-
cidadeOrigem.latitude)*math.Pi/180)*math.Cos((cidadeOrigem.longitude-
cidadeDestino.longitude)*math.Pi/180))

}

func getArrayOfCities(fileCities *bufio.Reader) []city {

	line, err := readLine(fileCities)
	var cities = []city{}
	for err == nil {
		cities = addCity(cities, line)
		line, err = readLine(fileCities)
	}

	rand.Seed(time.Now().UnixNano())
	cities = shuffle(cities)

	return cities
}

func shuffle(cities []city) []city {
	for index := range cities {
		rand := rand.Intn(index + 1)
		cities[index], cities[rand] = cities[rand], cities[index]
	}
	return cities
}

func addCity(cities []city, line string) []city {
	id, latitude, longitude := convertLineOfCity(line)

	cidade := city{id: id, latitude: latitude, longitude: longitude}
	cities = append(cities, cidade)
	return cities
}

func convertLineOfCity(line string) (id int, latitude float64, longitude
float64) {

	var err error
```

```
	idString := strings.Split(line, " ")[0]
	latitudeString := strings.Split(line, " ")[1]
	longitudeString := strings.Split(line, " ")[2]

	id, err = strconv.Atoi(idString)
	latitude, err = strconv.ParseFloat(latitudeString, 64)
	longitude, err = strconv.ParseFloat(longitudeString, 64)

	if err != nil {
		fmt.Printf("Erro na conversao de uma das linhas das cidades:
%v\n", err)
		os.Exit(1)
	}

	return id, latitude, longitude
}

func readCity(fileDirectory string) *bufio.Reader {
	fileCities, err := os.Open(fileDirectory)
	if err != nil {
		fmt.Printf("Erro ao abrir o Arquivo: %v\n", err)
		os.Exit(1)
	}
	return bufio.NewReader(fileCities)
}

func readLine(r *bufio.Reader) (string, error) {
	var (
		isPrefix bool  = true
		err      error = nil
		line, ln []byte
	)
	for isPrefix && err == nil {
		line, isPrefix, err = r.ReadLine()
		ln = append(ln, line...)
	}
	return string(ln), err
}
```

Anexo III – Programa TESTE Paralelo

```
package main

import (
	"bufio"
	"fmt"
	"math"
	"math/rand"
	"os"
	"runtime"
	"sort"
	"strconv"
	"strings"
	"time"

	"github.com/pmylund/sortutil"
)

type city struct {
	id                  int
	latitude, longitude float64
}

type chromosome struct {
	fitness float64
	cities  []city
}

type set map[interface{}]bool

func main() {

	runtime.GOMAXPROCS(runtime.NumCPU())

	if len(os.Args) > 4 {
		fileDirectory, populationSize, generations, mutation :=
readArgs()
		searchInstance(fileDirectory, populationSize, generations,
mutation)

	} else {
		fmt.Println("Passe os argumentos para executar o experimento")
		fmt.Println("ARGS: fileDirectory, populationSize, generations,
mutation")
	}

}

func readArgs() (string, string, string, string) {
	return os.Args[1], os.Args[2], os.Args[3], os.Args[4]
}

func searchInstance(fileDirectory, populationSizeString,
generationsString, mutationString string) {
```

```
	populationSize, err := strconv.Atoi(populationSizeString)
	if err != nil {
		fmt.Println("O tamanho da polulação é inválido\n", err)
		os.Exit(1)
	}
	generations, err := strconv.Atoi(generationsString)
	if err != nil {
		fmt.Println("O valor das gerações é inválido\n", err)
		os.Exit(1)
	}
	mutation, err := strconv.ParseFloat(mutationString, 64)
	if err != nil {
		fmt.Println("O valor das mutações é inválido\n", err)
		os.Exit(1)
	}

	fileCities := readCity(fileDirectory)
	cities := getArrayOfCities(fileCities)
	var elitismCut = (populationSize * 25) / 100

	if elitismCut%2 != 0 {
		elitismCut++
	}

	var population []chromosome
	populationChan := make(chan chromosome, populationSize)

	for index := 0; index < populationSize; index++ {
		go createChromosomeOfInitialPopulation(populationChan, cities)
	}

	for index := 0; index < populationSize; index++ {
		population = append(population, <-populationChan)
	}

	for index := 0; index < generations; index++ {
		sortutil.AscByField(population, "fitness")

		population := elitism(populationSize, elitismCut, population)

		maxFilhos := populationSize - len(population)

		newPopulationChan := make(chan []city, maxFilhos)

		for index := 0; index < maxFilhos; index++ {

			var indexes = randomInts(2, 0, populationSize,
makeRandomNumberGenerator())
			sort.Ints(indexes)

			go ox(newPopulationChan,
population[rand.Intn(elitismCut)].cities,
population[rand.Intn(elitismCut-1)].cities, indexes[0], indexes[1])

		}
```

```
            for index := 0; index < maxFilhos; index++ {

                  newcro := make(chan chromosome)
                  go createChromosome(newcro, <-newPopulationChan)

                  population = append(population, mutate(<-newcro,
populationSize, mutation))

            }
            fmt.Println(population[0].fitness)
      }

      fmt.Println(population[0].fitness)

}

func ox(newGen chan []city, pai1, pai2 []city, corte1, corte2b int) {
      var (
            n  = len(pai1)
            o1 = make([]city, n)
            o2 = make([]city, n)
      )

      copy(o1[corte1:corte2b], pai1[corte1:corte2b])
      copy(o2[corte1:corte2b], pai2[corte1:corte2b])

      var o1Lookup, o2Lookup = make(set), make(set)
      for index := corte1; index < corte2b; index++ {
            o1Lookup[pai1[index]] = true
            o2Lookup[pai2[index]] = true
      }

      var j1, j2 = corte2b, corte2b
      for index := corte2b; index < corte2b+n; index++ {
            var k = index % n
            if !o1Lookup[pai2[k]] {
                  o1[j1%n] = pai2[k]
                  j1++
            }
            if !o2Lookup[pai1[k]] {
                  o2[j2%n] = pai1[k]
                  j2++
            }
      }

      newGen <- o1
}

func mutate(gene chromosome, populationSize int, motation float64)
chromosome {
      if motation == 0 {
            return gene
      }

      for rand.Float64() < motation {
            position1 := rand.Intn(populationSize)
```

```
            position2 := rand.Intn(populationSize)

            aux := gene.cities[position1]
            gene.cities[position1] = gene.cities[position2]
            gene.cities[position2] = aux

      }

      return gene
}

func makeRandomNumberGenerator() *rand.Rand {
      return rand.New(rand.NewSource(time.Now().UnixNano()))
}

func randomInts(k, min, max int, rng *rand.Rand) (ints []int) {
      ints = make([]int, k)
      for i := 0; i < k; i++ {
            ints[i] = i + min
      }
      for i := k; i < max-min; i++ {
            var j = rng.Intn(i + 1)
            if j < k {
                  ints[j] = i + min
            }
      }
      return
}

func elitism(populationSize int, percent int, geracao []chromosome)
[]chromosome {

      return geracao[0:percent]
}

func createChromosome(chos chan chromosome, cities []city) {
      calculateFitnessChan := make(chan float64)
      go calculateFitness(calculateFitnessChan, cities)
      chos <- chromosome{fitness: <-calculateFitnessChan, cities: cities}
}

func createChromosomeOfInitialPopulation(populationChan chan chromosome,
cities []city) {

      tmp := make([]city, len(cities))
      copy(tmp, cities)

      individuo := shuffle(tmp)
      calculateFitnessChan := make(chan float64)
      go calculateFitness(calculateFitnessChan, individuo)
      populationChan <- chromosome{fitness: <-calculateFitnessChan,
cities: individuo}
}

func calculateFitness(calculateFitnessChan chan float64, cities []city)
{
```

```
	var length = len(cities) - 1
	var fitness float64

	for index := 0; index < length; index++ {
		fitness += calculateDistanceCoordenate(cities[index],
cities[index+1])
	}
	fitness += calculateDistanceCoordenate(cities[length], cities[0])

	calculateFitnessChan <- fitness

}

func calculateDistanceCoordenate(cidadeOrigem, cidadeDestino city)
float64 {

	return 6371 * math.Acos(math.Cos(math.Pi*(90-
cidadeDestino.latitude)/180)*math.Cos((90-
cidadeOrigem.latitude)*math.Pi/180)+math.Sin((90-
cidadeDestino.latitude)*math.Pi/180)*math.Sin((90-
cidadeOrigem.latitude)*math.Pi/180)*math.Cos((cidadeOrigem.longitude-
cidadeDestino.longitude)*math.Pi/180))

}

func readCity(fileDirectory string) *bufio.Reader {
	fileCities, err := os.Open(fileDirectory)
	if err != nil {
		fmt.Printf("Erro ao abrir o Arquivo: %v\n", err)
		os.Exit(1)
	}
	return bufio.NewReader(fileCities)
}

func getArrayOfCities(fileCities *bufio.Reader) []city {

	line, err := readLine(fileCities)
	var cities = []city{}
	for err == nil {
		cities = addCity(cities, line)
		line, err = readLine(fileCities)
	}

	return cities
}

func readLine(r *bufio.Reader) (string, error) {
	var (
		isPrefix bool  = true
		err      error = nil
		line, ln []byte
	)
	for isPrefix && err == nil {
		line, isPrefix, err = r.ReadLine()
		ln = append(ln, line...)
```

```
	}
	return string(ln), err
}

func addCity(cities []city, line string) []city {
	id, latitude, longitude := convertLineOfCity(line)

	cidade := city{id: id, latitude: latitude, longitude: longitude}
	cities = append(cities, cidade)
	return cities
}

func convertLineOfCity(line string) (id int, latitude float64, longitude
float64) {

	var err error

	idString := strings.Split(line, " ")[0]
	latitudeString := strings.Split(line, " ")[1]
	longitudeString := strings.Split(line, " ")[2]

	id, err = strconv.Atoi(idString)
	latitude, err = strconv.ParseFloat(latitudeString, 64)
	longitude, err = strconv.ParseFloat(longitudeString, 64)

	if err != nil {
		fmt.Printf("Erro na conversao de uma das linhas das cidades:
%v\n", err)
		os.Exit(1)
	}

	return id, latitude, longitude
}

func shuffle(cities []city) []city {
	for index := range cities {
		rand := rand.Intn(index + 1)
		cities[index], cities[rand] = cities[rand], cities[index]
	}
	return cities
}
```

Anexo IV – Script base para execução automatizada dos testes

```
#!/bin/bash

for ((cont=0;cont<31;cont++))
do
      /usr/bin/time -p -a -o /home/luizalexandrew/Developer/Github/AG-
CaixeiroViajante/GO/paralelo/resultadoParalelo123.out
 /home/luizalexandrew/Developer/Github/AG-
CaixeiroViajante/GO/paralelo/main cidades123.bs 100 1000 0.1
done
```