
PROBLEMA DO CAIXEIRO- VIAJANTE

Eduardo Savian, Marcos Fehlaue

INTRODUÇÃO

- O problema do caixeiro-viajante (PCV) é um problema que tenta determinar a menor rota para percorrer uma série de cidades (visitando uma única vez cada uma delas), retornando à cidade de origem.
- Ele é um problema de otimização NP-difícil
- O método dos algoritmos genéticos é utilizado devido à simplicidade de operação, eficácia pela determinação de um máximo global e aplicabilidade em problemas onde se desconhece o modelo matemático.

PONTOS POSITOS E NEGATIVOS

- Bons em explorar uma vasta área do espaço de busca;
- Ele não ficam facilmente presos em mínimos locais;
- Combina boas soluções existentes para encontrar melhores soluções;
- Pode ser lento para convergir para uma solução ótima;
- Não garantem encontrar a solução ótima;
- Pode ser computacionalmente caro, especialmente se forem necessárias muitas gerações para convergir para uma solução satisfatória.

ENCONTRAR MELHOR ROTA

```
func findBestRoute(matrix [][]int, population [][]int) ([]int, int) {
    bestRoute := make([]int, len(population[0]))
    bestDistance := int(^uint(0) >> 1)

    for _, route := range population {
        distance := calculateTotalDistance(matrix, route)
        if distance < bestDistance {
            bestDistance = distance
            copy(bestRoute, route)
        }
    }

    return bestRoute, bestDistance
}
```

CALCULAR DISTÂNCIA TOTAL

```
func calculateTotalDistance(matrix [][]int, route []int) int {  
    totalDistance := 0  
    numCities := len(route)  
  
    for i := 0; i < numCities-1; i++ {  
        totalDistance += matrix[route[i]][route[i+1]]  
    }  
  
    totalDistance += matrix[route[numCities-1]][route[0]]  
  
    return totalDistance  
}
```

ALGORITMO GENÉTICO

```
func geneticAlgorithm(matrix [][]int, numGenerations int, populationSize int) ([]int, int) {  
    rand.New(rand.NewSource(time.Now().UnixNano()))  
  
    numCities := len(matrix)  
    population := initializePopulation(numCities, populationSize)  
    bestRoute := make([]int, numCities)  
    bestDistance := int(^uint(0) >> 1)
```

ALGORITMO GENÉTICO

```
for generation := 0; generation < numGenerations; generation++ {  
    fmt.Print()  
    population = evaluateAndSelect(matrix, population)  
    population = crossoverAndMutate(population)  
    bestInGeneration, bestDistInGeneration := findBestRoute(matrix, population)  
  
    if bestDistInGeneration < bestDistance {  
        bestDistance = bestDistInGeneration  
        copy(bestRoute, bestInGeneration)  
    }  
}  
  
return bestRoute, bestDistance  
}
```

INICIAR POPULAÇÃO

```
func initializePopulation(numCities, populationSize int) [][]int {  
    population := make([][]int, populationSize)  
    for i := 0; i < populationSize; i++ {  
        route := rand.Perm(numCities)  
        population[i] = route  
    }  
    return population  
}
```

AVALIAR E SELECIONAR

```
func evaluateAndSelect(matrix [][]int, population [][]int) [][]int {  
    populationSize := len(population)  
    fitness := make([]int, populationSize)  
  
    for i, route := range population {  
        fitness[i] = calculateTotalDistance(matrix, route)  
    }  
}
```

AVALIAR E SELECIONAR

```
selectedPopulation := make([][]int, populationSize/2)
for i := 0; i < populationSize/2; i++ {
    bestIdx := 0
    for j := 1; j < populationSize; j++ {
        if fitness[j] < fitness[bestIdx] {
            bestIdx = j
        }
    }
    selectedPopulation[i] = population[bestIdx]
    fitness[bestIdx] = int(^uint(0) >> 1)
}

return selectedPopulation
}
```

CRUZAR E MUTAR

```
func crossoverAndMutate(population [][]int) [][]int {
    populationSize := len(population)
    newPopulation := make([][]int, populationSize*2)

    for i := 0; i < populationSize; i++ {
        parent1 := population[rand.Intn(populationSize)]
        parent2 := population[rand.Intn(populationSize)]
        child := crossover(parent1, parent2)
        mutate(child)
        newPopulation[i] = parent1
        newPopulation[populationSize+i] = child
    }

    return newPopulation
}
```

CRUZAR

```
func crossover(parent1, parent2 []int) []int {  
    numCities := len(parent1)  
    child := make([]int, numCities)  
    copy(child, parent1)  
    start, end := rand.Intn(numCities), rand.Intn(numCities)  
    if start > end {  
        start, end = end, start  
    }  
  
    childPart := make(map[int]bool)  
    for i := start; i <= end; i++ {  
        childPart[child[i]] = true  
    }  
}
```

CRUZAR

```
idx := 0
for i := 0; i < numCities; i++ {
    if !childPart[parent2[i]] {
        for idx >= start && idx <= end {
            idx++
        }
        child[idx] = parent2[i]
        idx++
    }
}

return child
}
```

MUTAR

```
func mutate(route []int) {  
    numCities := len(route)  
    if rand.Float64() < 0.4 {  
        i, j := rand.Intn(numCities), rand.Intn(numCities)  
        route[i], route[j] = route[j], route[i]  
    }  
}
```

REFERÊNCIAS BIBLIOGRÁFICAS

- SHENDY, R. **Traveling Salesman Problem (TSP) using Genetic Algorithm (Python)**. Disponível em: <<https://medium.com/aimonks/traveling-salesman-problem-tsp-using-genetic-algorithm-fea640713758>>.
- **Traveling Salesman Problem using Genetic Algorithm**. Disponível em: <<https://www.geeksforgeeks.org/traveling-salesman-problem-using-genetic-algorithm/>>.
- WIKIPEDIA CONTRIBUTORS. **Genetic algorithm**. Disponível em: <https://en.wikipedia.org/wiki/Genetic_algorithm>.
- WIKIPEDIA CONTRIBUTORS. **Travelling salesman problem**. Disponível em: <https://en.wikipedia.org/w/index.php?title=Travelling_salesman_problem&oldid=1225477054>.

OBRIGADO