



Análise de algoritmos

O CAIXEIRO-VIAJANTE

Baseado no livro Entendendo Algoritmos, Um guia ilustrado para programadores e outros curiosos. Aditya Y. Bhargava, Novatec, 2017.





Análise de algoritmos

O CAIXEIRO-VIAJANTE

O Problema do Caixeiro Viajante (PCV) possui complexidade de otimização combinatória. A ideia é simular o menor trecho possível a ser percorrido, analisando as possibilidades que devem ser visitadas e ao final, retornar ao ponto de partida, visitando cada parada somente uma vez.

O tempo de execução na notação Big O para o problema do caixeiro-viajante é de $O(n!)$ ou tempo fatorial e performa melhor para menores números.

De maneira geral, não existe um algoritmo polinomial que resolva de forma ótima todos os casos, mas ao escolher a forma de se resolver, é necessário considerar o tamanho do problema, precisão desejada, recursos computacionais e características da aplicação. Para solucionar esse problema, faremos uso do algoritmo heurístico da **Inserção Mais Próxima**, um método que é simples e rápido, mas não garante a solução ótima.





Análise de algoritmos

O CAIXEIRO-VIAJANTE

À título de curiosidade, as formas de resolvermos esse problema, são por meio de:

- Algoritmos exatos (programação dinâmica, ramificação e corte);
- Algoritmos heurísticos (Inserção mais Próxima, Heurística 2-opt e algoritmos de Colônia de Formigas) ou via;
- Algoritmos metaheurísticos (Simulated Annealing, busca tabu).

APLICAÇÕES

Em geral, encontra-se em problemas logísticos e roteirização de viagens.

- Logística e distribuição: otimização nas rotas de entrega, coleta de lixo;
- Manufatura e produção: otimização em layout de fábrica, tarefas em linha de produção;
- Varejo e vendas: otimização de rotas de vendas bem como planejamento de merchandising;
- Turismo: roteirização de viagens e otimização de passeios turísticos.





CAIXEIRO-VIAJANTE



```
package main

import (
    "fmt"
    "math"
)

type city struct {
    Name      string `json:"name" `
    Latitude   float64 `json:"latitude" `
    Longitude  float64 `json:"longitude" `
    Visited    bool    `json:"visited" `
}

type route struct {
    Cities    []city `json:"cities" `
    Distance  float64 `json:"distance" `
}
```





CAIXEIRO-VIAJANTE



```
type citiesRepository interface {  
    getAll() ([]city, error)  
}  
  
type previousCitiesRepository interface {  
    cities []city  
}  
  
func (repo *previousCitiesRepository) getAll() ([]city, error) {  
    return repo.cities, nil  
}  
  
type findRouteUserCase interface {  
    execute(cities []city) (route, error)  
}  
  
type nearestInsetionHeuristic struct {  
    repository citiesRepository  
}
```





CAIXEIRO-VIAJANTE



```
func (n *nearestInsertionHeuristic) execute(cities []city) (route, error) {  
    if len(cities) < 2 {  
        return route{}, fmt.Errorf("minimum two cities required")  
    }  
    for i := range cities {  
        cities[i].Visited = false  
    }  
    currentCity := cities[0]  
    currentCity.visited = true  
    remainingCities := cities[1:]  
  
    route := route{Cities: []city{currentCity}}  
    for len(remainingCities) > 0 {  
        var nearestCityIndex int  
        nearestDistance := math.MaxFloat64
```





CAIXEIRO-VIAJANTE



```
for i, c := range remainingCities {
    if !c.Visited && distance(currentCity, c) < nearestDistance {
        nearestCityIndex = i
        nearestDistance = distance(currentCity, c)
    }
}
currentCity = remainingCities[nearestCityIndex]
currentCity.Visited = true
route.Cities = append(route.Cities, currentCity)
route.Distance += nearestDistance
remainingCities = append(remainingCities[:nearestCityIndex],
remainingCities[nearestCityIndex+1:]...)
}
route.Distance += distance(route.Cities[len(route.Cities)-1], route.Cities[0])
return route, nil
}
```





CAIXEIRO-VIAJANTE



```
func distance(firstCity city, secondCity city) float64 {  
    return math.Sqrt(math.Pow(firstCity.Longitude-  
secondCity.Longitude, 2) + math.Pow(firstCity.Latitude-  
secondCity.Latitude, 2))  
}  
  
func main() {  
    cities := []city{  
        {Name: "Mata Grande", Latitude: -9.118243, Longitude: -37.732298},  
        {Name: "Presidente Figueiredo", Latitude: -2.029813, Longitude:  
-60.02338},  
        {Name: "Gramado dos Loureiros", Latitude: -27.442895, Longitude:  
-52.91486},  
        {...},  
    }  
}
```





CAIXEIRO-VIAJANTE



```
repository := previousCitiesRepository{cities: cities}
userCase := nearestInsertionHeuristic{repository: &repository}

route, err := userCase.execute(cities)
if err != nil {
    fmt.Println("error finding route:", err)
    return
}

fmt.Println("route:", route)
for _, city := range route.Cities {
    fmt.Println("-", city.Name)
}
fmt.Println("total distance:", route.Distance)
}
```





SAÍDA DO PROGRAMA



CENÁRIO OTIMISTA

Route: [{Mata Grande -9.118243 -37.732298 true} {Casa Nova -9.164083 -40.9739986162 true} {Irupi -20.350122 -41.644359 true} {Gaspar -26.933597 -48.953428 true} {Gramado dos Loureiros -27.442895 -52.91486 true} {Presidente Figueiredo -2.029813 -60.02338 true}] 105.15682219482709}

- Mata Grande
- Casa Nova
- Pentecoste
- Cândido Mendes
- Campos Belos
- Indiará
- Irupi
- Gaspar
- Gramado dos Loureiros
- Presidente Figueiredo

total distance: 105.15682219482709





Análise de algoritmos

CONSIDERAÇÕES

Para o exemplo apresentado, seguiu-se as boas práticas de desenvolvimento com *clean architecture*, *clean code* e SOLID, na tentativa de simular um cenário otimista.

Essa iniciativa vai de encontro com a ideia de trazer conteúdos relevantes altamente abordados em processos seletivos e desmitificar a ideia de algoritmos e estrutura de dados. Espero que seja de bom proveito e bons estudos.