Universidade Federal Rural de Pernambuco

Departamento de Estatística e Informática

Bacharelado em Sistemas de Informação

FLYFOOD

José Carlos Eliodoro de Santana

Recife

12 de 2022

1. Introdução

1.1 Apresentação e Motivação

Num contexto no qual os sistemas de distribuição inseridos nos ambientes urbanos, os fornecedores de serviços tendem enfrentar diversos desafios no seu dia-a-dia, como incertezas relacionadas com a entrega das mercadorias, principalmente com o alto custo da mão-de-obra humana, a implementação do uso de meios alternativos de vem sendo cada vez requisitada no ramo alimentício.

Tendo em vista essas necessidades viram que a utilização de drones, seria uma solução viável contanto que possuíssem um algoritmo de roteamento eficiente no qual os drones pudessem operar, com isso em mente o intuito deste trabalho propor um algoritmo de roteamento de drones com a finalidade de fornecer um algoritmo que trace os menores percursos tendo em vista reduzir os custos e concluir todas as entregas, dentro do ciclo da bateria do drone.

Temos como justificativa para o trabalho a necessidade de reduzir custos como a mão-de-obra escassa, melhorar o tempo de entrega das mercadorias, proporcionando assim uma diminuição do tempo de entrega dos produtos para os clientes.

1.2 Formulação do problema

O problema pode ser formalizado matematicamente como um problema de otimização de rotas.

Entrada desse problema seria uma matriz que representa a distância entre os pontos de entrega, onde cada elemento da matriz representa a distância entre dois pontos. Além disso, também seria necessário informar a posição do ponto de origem e retorno (R) na matriz.

A função objetivo seria minimizar a soma das distâncias percorridas pelo drone entre os pontos de entrega e o ponto de origem e retorno, considerando que ele só consegue se mover na horizontal ou na vertical. Matematicamente, isso pode ser representado por:

minimize ($\Sigma d(i, j)$): onde $d(i, j)$ é a distância percorrida pelo drone entre os pontos i e j, e Σ é a soma de todas as distâncias percorridas pelo drone.

Os pontos de entrega e o ponto de origem e retorno podem ser representados como vértices em um grafo, onde cada aresta representa uma distância $d(i, j)$ entre os vértices. O algoritmo deve encontrar o caminho de menor custo no grafo, ou seja, o caminho que minimiza a função objetivo acima.

A restrição é que o drone só se move na horizontal ou na vertical, o que pode ser representado por uma restrição de movimento no grafo, onde cada vértice só tem arestas com vértices vizinhos na horizontal ou na vertical.

1.3 Objetivos

Este trabalho tem como objetivo principal desenvolver um algoritmo que possa calcular a menor distância entre os pontos de uma matriz, representando pontos em uma cidade, levando em consideração os limites impostos para o movimento dos drones. O objetivo é encontrar uma forma de otimizar o transporte de mercadorias, reduzindo os custos envolvidos.

Para alcançar esse objetivo, será necessário realizar uma pesquisa sobre os algoritmos existentes para o cálculo de menor distância e suas limitações, bem como sobre os limites de movimento dos drones. Com base nessa pesquisa, será possível desenvolver um algoritmo que atenda às necessidades específicas deste trabalho. O algoritmo desenvolvido será testado e avaliado com dados reais de uma cidade, com o objetivo de verificar a sua eficácia na redução de custos de transporte de mercadorias.

Além disso, serão propostas possíveis melhorias e aplicações futuras do algoritmo desenvolvido. Espera-se que este trabalho possa contribuir para a otimização do transporte de mercadorias, proporcionando benefícios econômicos e ambientais.

2.Referencial Teórico

Nesta seção, serão apresentadas as referências utilizadas para a elaboração do trabalho, a fim de proporcionar uma melhor compreensão dos temas abordados.

2.1 O Problema do Caixeiro Viajante

O problema do caixeiro viajante(PCV) é um clássico exemplo de problemas de otimização combinatória. Tendo em vista que pela sua fácil compreensão ele é utilizado, para solucionar vários problemas da sua ampla classe de problemas que são os NP-completo, ele é utilizado em diversas áreas de estudos tal como na utilização de roteamento de veículos, problema de estoque de depósitos, nos raios-x, criptografia, etc.

O problema do caixeiro viajante é um é baseado no ciclo hamiltoniano, no qual é um circuito no grafo hamiltoniano G = (V, E) o qual ele visita todos os vértices exclusivamente uma vez a fim de encontrar o caminho menos custoso. Deste moto um vendedor que deseja viajar para N cidades do qual ele só poderá visitar cada uma única vez, nesse caso existiram pela

propriedade da permutação haverá (N-1)! Rotas possíveis sendo uma das cidades o ponto de partida e de retorno. Tanto o problema do caixeiro viajante quanto o problema exposto no trabalho possuem uma relação, ambos buscam visitar todos os pontos mapeados na rota menos custosa. Logo, ambos se utilizam da mesma teoria para elaborar uma solução coerente com a sua realidade.

2.2 Complexidade de Algoritmo

O custo computacional é definido como a quantidade de recursos, tais como o tempo de processamento e espaço de armazenamento da maquina, o que faz com que um mesmo algoritmo tenha um desempenho diferente em maquinas com hardwares diferentes.

2.3 Classes de problemas

Na Teoria da Complexidade Computacional, uma Classe de problema é um conjunto de problemas relacionados aos recursos computacionais baseados em complexidade, das quais podemos dividi-los em quatro classes de problemas que seriam P, NP, NP-completo e NP-difícil.

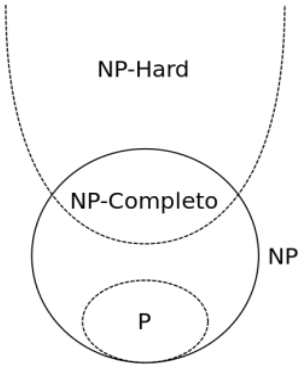


Diagrama de Euler para o conjunto de problemas P, NP, NP-completo, e NP-hard. Imagem da Wikipédia .

Classe P(Polynomial): Na classe P, estão os algoritmos que podem ser resolvidos em tempo polinomial, ou seja, o tempo de pior caso e de $O(n^k)$, um exemplo são problemas de decisão.

Classe NP(Nondeterministic polynomial): Na classe de NP, estão os problemas que são em tempo polinomial por uma máquina de Turing não-determinística, nessa classe estão problemas de busca e otimização.

Classe NP-completo: É um subconjunto de NP e São problemas NP que são "mais difíceis" do que outros problemas NP, ou seja, é possível transformar qualquer outro problema NP em um problema NP-completo em tempo polinomial.

Classe NP-difícil: São problemas NP para os quais não existe um algoritmo conhecido que possa resolvê-los em tempo razoável. Os problemas NP-difíceis são tão difíceis quanto os problemas NP-completos.

1. Metodologia

Problema do Caixeiro Viajante (PCV) é o problema de passar por um conjunto de cidades (visitando cada cidade exatamente uma vez) e determinar a rota mais curta de volta à cidade original. Este é um problema de classe NP-difícil.

Os problemas NP-difícil é uma classe de problemas que são, pelo menos tão difíceis quanto os problemas mais difíceis em NP, sendo problemas que não são solucionando em tempo polinomial numa máquina de Turing não-determinística.

O PVC possui inúmeras aplicações no mundo real como o roteamento de veículos, no problema do flyfood, em sistemas de armazenamento, fiação de computadores, etc. Baseado no ciclo hamiltoniano no qual é um circuito no grafo hamiltoniano, o qual visita todos os vértices do grafo apenas uma vez e por fim retorna a seu local de partida.

Na construção da solução do problema, desenvolvemos um método que busca pela menor roda por meio de um algoritmo que percorre por força bruta todas as possibilidades de rotas possíveis, a fim de encontrar a de menor custo.

A estrutura do algoritmo de força bruta se estabelece pela chamada de um função inicial chamada menor_dist(url) com o parâmetro contendo o caminho do arquivo o qual ele vai ler uma matriz preestabelecida pelo usuário, os pontos de entrega ou vértices serão a representação das letras contidas na matriz de entrada, e criará uma classe com um método para ler e outro para criar uma rota inicial contendo todos os pontos que será posto no grafo com a eliminação da rota inicial que consequentemente também é a final do grafo a qual foi estabelecida na matriz como ‘R’ ele faz uma permutação com os vertesses restantes pela função permut(V) é uma função que utiliza de backtracking para menor utilização computacional, onde V é um vetor contendo n-1 elementos definidos na matriz, essa função retorna um vetor contendo (n-1)! elementos, que são todas as rotas possíveis.

Após termos todas as rotas possíveis, a função menor_dist retorna a chamada da função menor(routes, R) onde routes são as rotas possíveis e R a rota de início, ele faz o cálculo da distância de todas as rotas iniciando em R e terminando nele mesmo e verifica a menor rota, após fazer todas as possibilidades possíveis ele retorna a menor rota e a distância percorrendo ela.

3.1 Pseudo algoritmo

Classe que representa um ponto da matriz.

```
1:  classe House():
2:      metodo_construtor(x, y:Inteiro, name :Caractere):
3:      inicio
4:          parametro.x <- x
5:          parametro.y <- y
6:          parametro.name <- name
7:      fimmetodo
8:      metodo distant(house :Objeto):
9:      inicio
10:         retorne valor_absoluto(parametro.x - house.x) + valor_absoluto(parametro.y - house.y)
11:     fimmetodo
12:Fimclasse
```

A complexidade temporal desta classe é O(1) porque o construtor e o método "distant" ambos realizam operações de tempo constante. O construtor atribui valores às variáveis da classe e o método "distant" realiza operações matemáticas simples nas variáveis da classe. Todas essas operações levam tempo constante, então a complexidade temporal geral é O(1).

Classe para entrada de dados.

```
1:  classe DateInput():
2:      metodo_construtor(url :Caractere)
3:      inicio
4:          parametro.url <- url
5:      fimmetodo
6:      metodo read_file(url:Caractere):
7:      var
8:          line :Vetor
9:      inicio
10:         abrir(parametro.url, 'r') chame obj:
11:             line <- obj.readlines()
12:         retorne line
13:     fimmetodo
14:     metodo create_router(text :Caractere):
15:     var
16:         x, y :Inteiro
17:         position, line, ln :Vetor
18:     inicio
19:         x <- 0
20:         para (line cada item text) faca
21:             ln <- line.replace('\n', '').split(' ')
22:             para (y de 0 ate sizeof(ln) passo 1) faca
23:                 se ln[y] != '0' entao
24:                     position.acrescentar(House(x, y, ln[y]))
25:                 fimse
26:             fimpara
```

```
27:          x <- x + 1
28:       fimpara
29:       return position
30:    fimmetodo
31:Fimclasse
```

A complexidade temporal desta função é O(n^2) onde n é o número de linhas no arquivo de entrada. Há dois loops aninhados nas linhas 20-28, o loop externo roda n vezes, onde n é o número de linhas no arquivo de entrada, e o loop interno roda m vezes, onde m é o número de elementos em uma linha. Cada iteração do loop interno leva um tempo constante para realizar as operações nas linhas 23-24, então a complexidade temporal geral é O(n*m) que pode ser simplificada para O(n^2) se n é aproximadamente igual a m.

Classe para permutar elementos de um array.

```
1:  funcao permut(elements, perm = [])
2:  var
3:  list_permutation :Vetor
4:  i :Inteiro
5:  inicio
6:     se (sizeof(elements) = 0) entao
7:        retorne [perm]
8:     senao:
9:        para (i de 0 ate sizeof(len(elements)) passo 1) faca
10:          new_element <- list((elements[:i])) + list((elements[(i+1):]))
11:          new_perm <- list((perm) + [elements[i]])
12:          list_permutation.extend(permut(new_element, new_perm))
13:       retorne list_permutation
14:    fimse
15:Fimfuncao
```

A complexidade temporal desta função é O(n!) onde n é o comprimento do vetor de entrada "elements". Isso ocorre porque a cada chamada da função, o vetor de entrada "elements" é reduzida em um caractere e a função faz n chamadas a si mesma, criando um total de n! chamadas recursivas. Cada chamada recursiva requer uma quantidade constante de tempo para realizar as operações nas linhas 10-12, portanto, a complexidade temporal geral é O(n!).

Função Intermediaria que retorna menor rota

```
1:  funcao (routes :Vetor, r: Objeto)
2:  var
3:     minim, x, y :inteiro
4:     rt :caractere
5:     route, distanci :Vetor
6:  inicio
7:     para (x de 0 ate sizeof(routes) passo 1) faca
8:        route <- ['R']
9:        distanci <- [r.distant(routes[x][0])]
10:       para (y de 0 ate sizeof(routes[x]) passo 1) faca
```

```
11:         route.acrescentar(routes[x][y].name)
12:         se (y < sizeof(routes[x])-1) entao
13:            distanci.acrescentar(routes[x][y].distant(routes[x][y+1]))
14:         fimse
15:      fimpara
16:      route.acrescentar('R')
17:      distanci.acrescentar(r.distant(routes[x][-1]))
18:      se (minim = 0 or minim > sum(distanci)) entao
19:         minim <- sum(distanci)
20:         rt <- route
21:      fimse
22:   fimpara
23:   retorne 'A menor rota é %s com exatos %d dronômetros'%(rt, minim)
24:Fimfuncao
```

A complexidade temporal desta função é O(n^2) onde n é o tamanho do vetor de entrada "routes". Isso ocorre porque há um laço aninhado nas linhas 7-22, onde o laço externo executa n vezes e o laço interno executa m vezes, onde m é o tamanho de "routes[x]". Cada iteração do laço interno leva um tempo constante para realizar as operações nas linhas 11-14 e 18-20, portanto, a complexidade temporal geral é O(n*m) que pode ser simplificada para O(n^2) se n for aproximadamente igual a m.

Função para ler o arquivo de entrada

```
16:funcao menor_distancia_file(url)
17:var
18:   datas, r :Objeto
19:   matriz, routes :Vetor
20:inicio
21:   datas = DateEntry(url)
22:   matriz = datas.create_matriz(datas.read_file())
23:   routes = datas.create_routes(matriz)
24:   r = list(filtro el: el.name esta em 'R', routes))[0]
25:   routes = permut(list(filtro el: el.name esta em 'R', routes)))
26:   retorne menor(routes, r)
27:fimfuncao
```

A complexidade temporal desta função pode resumir a complexidade de todo o algoritmo é O(n! + n^2) onde n é o número de elementos no arquivo de entrada.

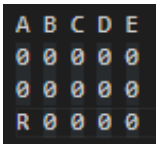
A primeira operação realizada é a leitura do arquivo, que leva O(n) tempo. Em seguida, as funções create_matriz e create_routes levam O(n) tempo cada. Em seguida, a função permut leva O(n!) tempo, pois gera todas as permutações possíveis da entrada. Por fim, a função menor tem complexidade O(n^2) devido ao laço aninhado nela..

Então, a complexidade temporal geral é O(n) + O(n) + O(n!) + O(n^2) que pode ser simplificada como O(n! + n^2).

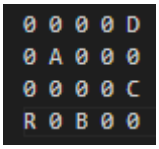
2. Experimentos

Nos experimentos foram inicialmente testados o caso base do problema, os resultados que foram obitidos apartir de 3 execuções de cada teste, o tempo exibido será em milissegundos da media de cada testes

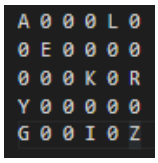
Resultado: A menor rota é ['R', 'A', 'D', 'C', 'B', 'R'] com exatos 14 dronômetros, tempo médio de 0,001000ms.



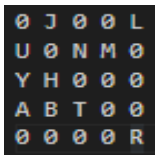
Resultado: A menor rota é ['R', 'A', 'B', 'C', 'D', 'E', 'R'] com exatos 14 dronômetros, tempo médio de 0,001545ms.



Resultado: A menor rota é ['R', 'L', 'K', 'E', 'A', 'Y', 'G', 'I', 'Z', 'R'] com exatos 22 dronômetros, tempo médio de 0,416512ms.



Resultado: A menor rota é ['R', 'L', 'M', 'N', 'J', 'U', 'Y', 'H', 'A', 'B', 'T', 'R'] com exatos 20 dronômetros, tempo médio de 46,883842ms.



3. Conclusão

Em resumo, o experimento realizado buscou avaliar a eficiência do algoritmo de força bruta para resolver o problema do caixeiro-viajante. O experimento foi realizado em uma máquina com um processador Intel Core i5 de 3.4 GHz, 8 GB de memória RAM e 128 GB de armazenamento em SSD, utilizando o sistema operacional Windows 10 Pro. O experimento foi realizado três vezes e foi utilizado a média dos tempos de execução para obter os resultados.

Os resultados mostraram que o algoritmo de força bruta foi capaz de encontrar a solução do problema do caixeiro-viajante, mas que o tempo de execução aumentou exponencialmente conforme o número de cidades aumentou. Para 4 cidades, o tempo de execução foi de 0.001ms, enquanto para 11 cidades foi de 592.89073ms. Esses resultados são consistentes com a complexidade do algoritmo, que é em ordem fatorial.

Em conclusão, o algoritmo de força bruta mostrou ser uma opção viável para problemas com um pequeno número de cidades, mas é ineficiente para problemas com um grande número de cidades. Dessa forma, futuros trabalhos devem explorar outras alternativas de algoritmos mais eficientes para resolver o problema do caixeiro-viajante.

Referências Bibliográficas

RODRIGUES, R. A. N.; CARVALHAES, M. F. A. Análise e Complexidade de Algoritmos: Backtracking e Força Bruta. **Revista Ada Lovelace**, *[S. l.]*, v. 1, p. 45–48,

2017. Disponível em: http://anais.unievangelica.edu.br/index.php/adalovelace/article/view/4117. Acesso em: 10 jan. 2023.

SILVEIRA, J.f. Porto da. Problema do Caixeiro Viajante. 2000. Disponível em: http://www.mat.ufrgs.br/~portosil/caixeiro.html.

Godoy Dotta, Alexandre (2021): USO DE PROGRAMAÇÃO EM 'R' APLICADO AO PROBLEMA CAIXEIRO VIAJANTE COM OFICINA PARA DEMOSTRAR DEMAIS APLICAÇÕES. figshare. Presentation. https://doi.org/10.6084/m9.figshare.13857299.v1