Relatório Matemática Discreta

Licenciatura em Engenharia Informática

ParkPulse Sofware

Prof. Ana Moura

Prof. Amélia Caldeira

Prof. Stella Abreu

Grupo 312 1NA/NB – CodeSawyers

André Azevedo - 1230932

Carlos Coelho - 1221808

Miguel Lopes - 1222183

Rui Margarido - 1230420

Tiago Sá – 1201925

Indice

Introdução

A gestão de espaços verdes exige a aplicação de ferramentas multidisciplinares para garantir um planeamento e gestão eficazes.

No início do planeamento de um parque, é crucial implementar sistemas de irrigação eficientes para os espaços verdes. Para isso, é realizado um levantamento topográfico que identifica as áreas ajardinadas/arbóreas e os caminhos existentes no parque. Com base nesse levantamento, o uso do algoritmo de Kruskal permite determinar os melhores percursos para conectar os diferentes pontos de irrigação de forma otimizada.

Na fase seguinte, são planeadas e implementadas medidas de segurança, incluindo a definição de um plano de emergência que abrange pontos de encontro e rotas de evacuação. Com o levantamento topográfico em mãos, um especialista em planeamento de emergências pode determinar os locais ideais para a instalação de sinais de emergência, designando-os como pontos de encontro. Estes pontos são estrategicamente localizados para serem visíveis e de fácil acesso, garantindo que todos saibam para onde se dirigir em situações de perigo. Através do algoritmo de Dijkstra, é possível identificar as rotas de evacuação mais eficientes no parque, orientando utilizadores e funcionários a partir de vários locais até os pontos de encontro de forma segura e rápida.

A análise de algoritmos é uma disciplina essencial na ciência da computação, pois permite avaliar e prever o desempenho de algoritmos em diferentes cenários. Uma das ferramentas fundamentais para essa análise é a notação "O", que nos ajuda a descrever a complexidade assintótica de um algoritmo em termos do tamanho de sua entrada.

O objetivo deste relatório é explorar a análise de complexidade de dois algoritmos distintos, utilizando a notação "O" como guia. Através dessa análise, iremos entender como o tamanho da entrada influencia o desempenho de cada algoritmo e comparar suas eficiências relativas.

O algoritmo de Kruskal é um algoritmo utilizado em ciência da computação para criar uma árvore geradora de custo mínimo em um grafo ponderado não direcionado. Uma árvore geradora de custo mínimo é um subgrafo que inclui todos os vértices do grafo original, conectando-os da forma mais eficiente possível, ou seja, com o menor custo total possível.

Inicialmente, para o algoritmo de Kruskal todas as arestas do grafo são ordenadas em ordem crescente de peso. De seguida, cada vértice é uma árvore isolada. Então, começa-se a selecionar as arestas em ordem crescente de custo e se uma aresta conecta dois vértices que estão em árvores diferentes, ela é adicionada à árvore geradora mínima, sendo que este processo é replicado até que todos os vértices estejam conectados em uma única árvore.

O algoritmo de Kruskal é extramente útil para a resolução de problemas que envolvem otimização de custos, visto que ajuda a encontrar a solução mais eficiente, garantindo a diminuição do custo total.

O algoritmo de Dijkstra é uma técnica usada em ciência da computação para encontrar o caminho mais curto entre dois pontos em um mapa, entre dois diferentes locais, considerando para isso as distâncias entre cada ponto e, assim, determinando a rota mais curta possível.

Ele é usado em muitas aplicações práticas onde encontrar a rota mais eficiente é de extrema importância.

US13

O algoritmo de Kruskal implementado é composto por 3 métodos principais:

1) runKruskalAlgorithm

Este é o método principal que executa o algoritmo de Kruskal. Primeiro, ele ordena as arestas pelo custo (sortEdges(edges)). Em seguida, ele inicia um array chamado "group" para armazenar a qual grupo cada vértice pertence. Após isso, percorre todas as arestas, selecionando-as adicionando à árvore geradora de custo mínimo.

2) *sortEdges*

Este método ordena as arestas pelo custo. Ele é crucial para garantir que as arestas sejam processadas em ordem crescente de custo pelo algoritmo de Kruskal.

3) findVertexIndex

Este método encontra o índice de um vértice na lista de vértices, itera sobre a lista de vértices para encontrar o índice do vértice fornecido e retorna esse índice.

1) Método: runKruskalAlgorithm

Pseudocódigo:

```
function runKruskalAlgorithm(vertices: list of string, edges: list of CSVLine) returns list of CSVLine
    sortEdges(edges) // Ordena as arestas pelo custo

    group := array[1..length(vertices)] of int // Array para armazenar a qual grupo cada vértice pertence
    for i from 1 to length(vertices) do // Inicializa o array de grupos
        group[i] := i // Inicialmente, cada vértice está em seu próprio grupo

    mst := [] // Inicializa a lista de arestas na árvore geradora mínima

    for each edge in edges do // Itera por todas as arestas em ordem crescente de custo
        group1 := group[findVertexIndex(edge.getX(), vertices)] // Encontra o grupo dos vértices de partida
        group2 := group[findVertexIndex(edge.getY(), vertices)] // Encontra o grupo dos vértices de destino

        if group1 ≠ group2 then // Verifica se os dois vértices estão no mesmo grupo para evitar ciclos e adiciona a aresta à árvore geradora mínima
            add edge to mst // Adiciona a aresta à árvore geradora mínima
            for k from 1 to length(group) do // Atualiza o grupo de todos os vértices no mesmo grupo que o vértice de destino
                if group[k] = group2 then
                    group[k] := group1

    return mst // Retorna a árvore geradora mínima
```

Cálculo da complexidade:

Linha	Código	Operações
2	sortEdges(edges) // Ordena as arestas pelo custo	C
4	group := array[1..length(vertices)] of int // Array para armazenar a qual grupo cada vértice pertence	A
5	for i from 1 to length(vertices) do // Inicializa o array de grupos	(n+1)A + (n+1)C
6	group[i] := i // Inicialmente, cada vértice está em seu próprio grupo	nA
8	mst := [] // Inicializa a lista de arestas na árvore geradora mínima	A
10	for each edge in edges do // Itera por todas as arestas em ordem crescente de custo	(n+1)A + (n+1)C
11	group1 := group[findVertexIndex(edge.getX(), vertices)] // Encontra o grupo dos vértices de partida	nC
12	group2 := group[findVertexIndex(edge.getY(), vertices)] // Encontra o grupo dos vértices de destino	nC
14	if group1 ≠ group2 then // Verifica se os dois vértices estão no mesmo grupo para evitar ciclos e adiciona a aresta à árvore geradora mínima	nC
15	add edge to mst // Adiciona a aresta à árvore geradora mínima	nA
16	for k from 1 to length(group) do // Atualiza o grupo de todos os vértices no mesmo grupo que o vértice de destino	n(n+1)A + n(n+1)C
17	if group[k] = group2 then	n*n*C
18	group[k] := group1	n*n*A
20	return mst // Retorna a árvore geradora mínima	R

$$Total = 2A + C + R + 2nA + 3nC + 2[(n+1)A + (n+1)C\,] + \; n^2A + n^2C + [n(n+1)(A+C)]$$

$$= 2 + 1 + 1 + 2n + 3n + 4n + 4 + n^2 + n^2 + 2n^2 + 2 = 9 + 9n + 4n^2$$

$$Estimativa\ O(n^2)$$

2) Método: sortEdges

Pseudocódigo:

```
1  procedure sortEdges(edges: list of CSVLine)
2      for i from 1 to length(edges) - 1 do
3          for j from 1 to length(edges) - i do
4              if edges[j].getCost() > edges[j + 1].getCost() then
5                  temporaryEdge := edges[j]
6                  edges[j] := edges[j + 1]
7                  edges[j + 1] := temporaryEdge
```

Cálculo da complexidade:

Linha	Código	Operações
2	for i from 1 to length(edges) - 1 do	(n+1)A + (n+1)C
3	for j from 1 to length(edges) - i do	n(n+1)A + n(n+1)C
4	if edges[j].getCost() > edges[j + 1].getCost() then	nnC
5	temporaryEdge := edges[j]	nnA
6	edges[j] := edges[j + 1]	nnA
7	edges[j + 1] := temporaryEdge	nnA

$$Total = (n+1)A + (n+1)C + n(n+1)(A+C) + 3\,n^2A + n^2C = 2n + 2 + 2n^2 + 2n + 3\,n^2 + n^2$$

$$= 2 + 4n + 3\,n^2 + 6n^2$$

$$Estimativa\ O(n^2)$$

3) Método: findVertexIndex

Pseudocódigo:

```
function findVertexIndex(vertex: string, vertices: list of string) returns int
    for i from 1 to length(vertices) do
        if vertices[i] = vertex then
            return i
    return -1
```

Cálculo da complexidade:

Linha	Código	Operações
2	for i from 1 to length(vertices) do	(n+1)A + (n+1)C
3	if vertices[i] = vertex then	nC
4	return i	R

$$Total = 3n + 3$$

$$Estimativa\ O(n)$$

US17 / US18

O algoritmo de Dijstra que desenvolvemos foi construído para responder diretamente a ambas as US 17 e 18 é composto pelos seguintes métodos:

1) Método: dijkstraCalculateShorterRoute

Calcula a rota mais curta a partir de cada vértice até ao ponto de encontro mais próximo.

2) Método Auxiliar: dijkstraAlgorithmRun

Executa o algoritmo de Dijkstra para calcular a rota mais curta a partir de um vértice inicial.

3) Método Auxiliar: findCloserAssemblyPoint

Encontra o ponto de encontro mais próximo de um conjunto de vértices.

4) Método Auxiliar: getRoute

Cada um desses métodos desempenha um papel crucial no cálculo das rotas mais curtas utilizando o algoritmo de Dijkstra, colaborando para produzir a lista final de rotas mais curtas a partir de cada vértice até o ponto de montagem mais próximo.

O algoritmo está implementado da seguinte forma:

1) *Método: dijkstraCalculateShorterRoute*

Pseudocódigo:

```
procedure dijkstraCalculateShorterRoute(costMatrix: array[1..n, 1..n] of int, assemblyPoints: array of int, verticeNames: array of string) returns list of Routes
    routes := [] // Inicializa a lista para armazenar as rotas finais
    counter := 0 // Contador para acompanhar o vértice atual

    for vertex in verticeNames do
        if counter not in assemblyPoints then // Se o vértice não é um ponto de reunião
            routeCost := array[1..n] of int // Inicializa o array para armazenar custos para alcançar cada vértice
            visitedVertices := array[1..n] of int // Inicializa o array para rastrear vértices visitados
            parent := array[1..n] of int // Inicializa o array para armazenar o pai de cada vértice no caminho mais curto

            // Executar o algoritmo de Dijkstra para o vértice atual
            call dijkstraAlgorithmRun(counter, costMatrix, routeCost, visitedVertices, parent)

            // Encontrar o ponto de reunião mais próximo do vértice atual
            closerAssemblyPoint := findCloserAssemblyPoint(assemblyPoints, routeCost, visitedVertices)

            if closerAssemblyPoint != -1 then // Se um ponto de reunião válido for encontrado
                // Obter a rota do vértice atual para o ponto de reunião mais próximo
                route := getRoute(counter, closerAssemblyPoint, parent, verticeNames)

        counter := counter + 1 // Passar para o próximo vértice

    return routes // Retornar a lista de rotas
```

Cálculo da complexidade:

Linha	Código	Operações
2	routes := []	A
3	counter := 0	A
5	for vertex in verticeNames do	(n+1)A + (n+1)C
6	if counter not in assemblyPoints:	nC
7	routeCost: array[1..n] of int	nA
8	visitedVertices := array[1..n] of int	nA
9	parent := array[1..n] of int	nA
12	call dijkstraAlgorithmRun(counter, costMatrix, routeCost, visitedVertices, parent)	nC
15	closerAssemblyPoint := findCloserAssemblyPoint(assemblyPoints, routeCost, visitedVertices)	nC
17	if closerAssemblyPoint != -1:	nC
19	route: getRoute(counter, closerAssemblyPoint, parent, verticeNames)	nC
21	counter counter + 1	nA + nOp
23	return routes	1R

$$Total = 2A + 1R + 4nA + 5nC + (n+1)A + (n+1)C = 2 + 1 + 4n + 5n + 2n + 2 = 11n + 5$$

$$Estimativa\ O(n)$$

2) Método Auxiliar: dijkstraAlgorithmRun

Pseudocódigo:

```
procedure dijkstraAlgorithmRun(counter: int, costMatrix: array[1..n, 1..n] of int, routeCost: array[1..n] of int, visitedVertices: array[1..n] of int, parent: array[1..n] of int)
   // Inicializar os arrays
   for i from 1 to length(routeCost) do
       routeCost[i] := MAX_VALUE // Definir todos os custos de rota como infinito
       visitedVertices[i] := 0 // Marcar todos os vértices como não visitados
       parent[i] := -1 // Definir todos os pais como -1 (sem pai)

   routeCost[counter] := 0 // Definir o custo para alcançar o vértice inicial como 0

   // Loop para processar cada vértice
   for i from 1 to length(routeCost) do
       closerVertex := -1
       closerCost := MAX_VALUE

       // Encontrar o vértice não visitado mais próximo
       for j from 1 to length(routeCost) do
           if visitedVertices[j] = 0 and routeCost[j] < closerCost then
               closerVertex := j
               closerCost := routeCost[j]

       // Se nenhum vértice mais próximo for encontrado, quebrar o loop
       if closerVertex = -1 then
           break

       visitedVertices[closerVertex] := 1 // Marcar o vértice mais próximo como visitado

       // Atualizar os custos e pais para os vizinhos do vértice mais próximo
       for j from 1 to length(routeCost) do
           if visitedVertices[j] = 0 and costMatrix[closerVertex][j] ≠ 0 and routeCost[closerVertex] ≠ MAX_VALUE and routeCost[closerVertex] + costMatrix[closerVertex][j] < routeCost[j] then
               routeCost[j] := routeCost[closerVertex] + costMatrix[closerVertex][j]
               parent[j] := closerVertex

```

Cálculo da complexidade:

Linha	Código	Operações
3	for i from 1 to length(routeCost) do	(n+1)A + (n+1)C
4	routeCost[i] := MAX_VALUE	nA
5	visitedVertices[i] := 0	nA
6	parent[i] := -1	nA
8	routeCost[counter] := 0	A
11	for i from 1 to length(routeCost) do	(n+1)A + (n+1)C
12	closerVertex := -1	nA
13	closerCost := MAX_VALUE	nA
16	for j from 1 to length(routeCost) do	n(n+1)A + n(n+1)C
17	if visitedVertices[j] = 0 and routeCost[j] < closerCost then	n*nC
18	closerVertex := j	n*n*A
19	closerCost := routeCost[j]	n*n*A
22	if closerVertex = -1 then break	nC
25	visitedVertices[closerVertex] := 1 // Marcar o vértice mais próximo como visitado	nA
28	for j from 1 to length(routeCost) do	n(n+1)A + n(n+1)C
29	if visitedVertices[j] = 0 and costMatrix[closerVertex][j] ≠ 0 and routeCost[closerVertex] ≠ MAX_VALUE and …	nC
30	routeCost[j] := routeCost[closerVertex] + costMatrix[closerVertex][j]	nA + nOp
31	parent[j] := closerVertex	nA

$$Total = 1A + 8nA + 2nC + 1nOp + 2n^2A + n^2C + 2[(n+1)A + (n+1)C] + 2[n(n+1)A + n(n+1)C]$$

$$= 1A + 8nA + 2nC + 1nOp + 2n^2A + n^2C + 2[(n+1)(A+C] + 2[n(n+1)(A+C)] = 1 + 8n + 2n + 1n + 2n^2 + n^2 + 2n + 2 + 4n(n+1)$$

$$= 3 + 17n + 7n^2 \qquad Estimativa\ O(n^2)$$

3) *Método Auxiliar: findCloserAssemblyPoint*

Pseudocódigo:

```
1  function findCloserAssemblyPoint(assemblyPoints: list of int, routeCost: array[1..n] of int, visitedVertices: array[1..n] of int) returns int
2     closerAssemblyPoint := -1
3     closerCost := MAX_VALUE
4
5     // Loop através de cada vértice
6     for i from 1 to length(routeCost) do
7
8         // Se o vértice é um ponto de reunião e foi visitado, e seu custo é o mais baixo encontrado até agora
9         if assemblyPoints.contains(i) and visitedVertices[i] = 1 and routeCost[i] < closerCost then
10            closerAssemblyPoint := i
11            closerCost := routeCost[i]
12
13    return closerAssemblyPoint // Retorna o ponto de reunião mais próximo
```

Cálculo da complexidade:

Linha	Código	Operações
1	closerAssemblyPoint := -1	A
2	closerCost := MAX_VALUE	A
6	for i from 1 to length(routeCost) do	(n+1)A + (n+1)C
9	if assemblyPoints.contains(i) and visitedVertices[i] = 1 and routeCost[i] < closerCost then	nC
10	closerAssemblyPoint := i	nA
11	closerCost := routeCost[i]	nA
13	return closerAssemblyPoint // Retorna o ponto de reunião mais próximo	R

$$Total = 2A + 2nA + 1nC + 1R + (n+1)A + (n+1)C = 5 + 5n$$

$$Estimativa\ O(n)$$

4) *Método Auxiliar: getRoute*

Pseudocódigo:

```
function getRoute(counter: int, closerAssemblyPoint: int, parent: array[1..n] of int, verticeNames: list of string) returns list of Routes
   route := [] // Inicializa a lista de rotas

   current := closerAssemblyPoint

   // Loop para construir a rota seguindo o array de pais
   while current ≠ -1 do
       route.add(new Routes(verticeNames[current], verticeNames[current], 0)) // Supondo que o custo não é necessário aqui
       current := parent[current] // Mover para o vértice pai


   return route // Retorna a rota construída
```

Cálculo da complexidade:

Linha	Código	Operações
2	route := [] // Inicializa a lista de rotas	A
4	current := closerAssemblyPoint	A
7	while current ≠ -1 do	(n+1)A + (n+1)C [algoritmo de procura linear]
8	route.add(new Routes(verticeNames[current], verticeNames[current], 0))	A
9	current := parent[current] // Mover para o vértice pai	A

$$Total = 2A + 2nA + 1nC + 1R + (n+1)A + (n+1)C = 5 + 5n$$

$$Estimativa\ O(n)$$

Conclusão

Resultados Kruskal:

Neste caso a complexidade é definida pelos métodos principais *sortEdges* e por tudo o que se passa dentro do *runKruskalAlgorithm*, não tendo em conta o método anterior, sendo o primeiro responsável pela ordenação das arestas e o segundo pela construção da solução. Após os cálculos verificamos que ambos apresentam complexidades $O(n^2)$ e quando duas partes de um algoritmo possuem a mesma complexidade, a complexidade total do algoritmo é simplesmente essa mesma ordem de grandeza. Esta análise indica que o algoritmo de Kruskal é eficiente para grafos com um número moderado de arestas e vértices, sendo adequado para a criação de uma árvore de caminhos mínimos em um contexto de gestão de emergências em parques.

Resultados Dijstra:

Neste caso temos o método *dijkstraAlgorithmRun* com complexidade $O(x^2)$ a ser chamado dentro do método *dijkstraCalculateShorterRoute* com complexidade $O(\mathrm{x})$, sendo que a complexidade final do algoritmo será determinada pelo produto das complexidades dos métodos resultando em $O(x^3)$. Esta complexidade cúbica pode ser um fator limitante em grafos muito grandes, afetando a escalabilidade e a eficiência do algoritmo em situações de grande número de vértices e arestas. No entanto, dada a sua precisão em encontrar os caminhos mais curtos, o algoritmo de Dijkstra continua a ser uma escolha importante para garantir rotas de evacuação eficientes e seguras.

A avaliação da complexidade dos algoritmos de Kruskal e Dijkstra é crucial para a tomada de decisões sobre qual algoritmo utilizar em diferentes cenários de emergência. Enquanto o algoritmo de Kruskal oferece uma solução eficiente e direta para a construção de árvores de caminhos mínimos, o algoritmo de Dijkstra proporciona rotas de menor custo mais detalhadas, embora com maior custo computacional.

Em conclusão, a escolha do algoritmo deve ser guiada pelo equilíbrio entre a necessidade de precisão nas rotas de evacuação e a eficiência computacional, levando em consideração o tamanho e a complexidade do parque em questão. A implementação prática dos algoritmos, acompanhada de uma análise detalhada da complexidade, assegura que os planos de evacuação sejam tanto eficazes quanto viáveis, proporcionando segurança e confiabilidade aos usuários dos espaços verdes.