Logística para Organização de Viagens

Desenho de Algoritmos, LEIC, 2º Ano

Fábio Araújo de Sá, up202007658@fe.up.pt
Marcos William Ferreira Pinto, up201800177@fe.up.pt

Porto, Maio de 2022

Descrição do problema

Uma agência de viagens pretende organizar caminhos e conexões para os seus clientes da melhor forma possível. Para isso precisa de diferentes funcionalidades:

1. Grupos sem separação:
   1.1. Maximização do número de elementos do grupo a transportar;
   1.2. Minimização do número de transbordos da viagem;
2. Grupos com separação:
   2.1. Determinar um encaminhamento para um grupo;
   2.2. Alterar um encaminhamento existente caso a dimensão do grupo aumente;
   2.3. Determinar a dimensão máxima de um grupo e um encaminhando;
   2.4. Determinar quando o grupo completo se reuniria novamente no destino;
   2.5. Determinar o tempo máximo de espera que elementos do grupo pode esperar em paragens intermediárias;

Cenário 1.1 - Formalização

Input:

- O - Nó de Origem do percurso desejado
- D - Nó de Destino do percurso desejado
- Grafo G(V, E) em que:
  - V - Número de vértices do Grafo G
  - E - Conjunto de arestas do Grafo G em que, para i ∈ [0..N]:
    - E_i.origin - index do nó de origem da aresta
    - E_i.destiny - index do nó de destino da aresta
    - E_i.time - tempo, em horas, que a aresta demora a ser percorrida
    - E_i.capacity - capacidade, em quantidade de pessoas, da aresta

Output / Variáveis de decisão:

- C - número de máximo pessoas a viajar
- V' = {O, V'_1, …, V'_{K-1}, D} , lista ordenada de Vértices que formam o caminho de capacidade C.

Restrições:

- $\forall e \in E, e_{origin} \in [1, N] \land e_{destination} \in [1, N] \land e_{time} > 0 \land e_{capacity} \geq 0$
- $\forall v \in V', v \in [1, N]$
- $C = min(\{e.capacity | e \in E \land e.origin, e.destination \in V' \land V'_i = e.origin \Rightarrow V'_{i+1} = e.destination\})$

Objetivo: $max(C)$

Cenário 1.1 - Algoritmo

Baseia-se numa adaptação do **Algoritmo de Dijkstra** com utilização de um set<pair<int, int>> para ordenar as capacidades de cada nó. Implementado com uma Red Black Tree, o set garante uma menor complexidade temporal em operações comuns neste algoritmo (ordenação e remoção) quando comparado com outras estruturas de dados.

Calcula a capacidade máxima de todos os vértices em relação ao vértice dado como origem e às capacidades das arestas. Sempre que a capacidade do nó é inferior à soma da capacidade do antecessor com a capacidade da aresta, há atualização (remoção seguida de inserção) do par correspondente no set.

A capacidade máxima descoberta será o mínimo das capacidades das arestas que compõem o caminho.

```cpp
void Graph::case1_a(int origin, int destiny) {

    cout << "Maximum number of passengers for the trip" << endl;

    set<pair<int, int>> capacities;

    reset();
    for (int i = 1 ; i < nodes.size() ; i++) {
        capacities.insert(make_pair(0, i));
    }

    nodes[origin].capacity = INF;
    capacities.erase(make_pair(0, origin));
    capacities.insert(make_pair(INF, origin));

    while (!capacities.empty()) {

        int currentNodeIndex = capacities.begin()->second;
        int currentNodeCapacity = capacities.begin()->first;
        capacities.erase(capacities.begin());

        for (Edge e : nodes[currentNodeIndex].adjacent) {
            if (min(currentNodeCapacity, e.capacity) > nodes[e.dest].capacity) {
                int oldCapacity = nodes[e.dest].capacity;
                nodes[e.dest].capacity = min(currentNodeCapacity, e.capacity);
                nodes[e.dest].parent = currentNodeIndex;
                capacities.erase(make_pair(oldCapacity, e.dest));
                capacities.insert(make_pair(nodes[e.dest].capacity, e.dest));
            }
        }
    }
    showPathCase1(origin, destiny);
}
```

Cenário 1.1 - Complexidade

Seja V o número de Vértices e E o número de Arestas, onde E ≥ V:

$$\mathbf{T(V, E)} = O\ (\ V + V*\log(V) + V(\ \log(V) + E*2*\log(V))\ ) \approx \mathbf{O\ (\ V*E\ \log(V)\ )}$$

Reset dos atributos dos vértices

Inserção no set de capacidades

Percorrer todos os vértices

Apagar o vértice do set de capacidades

Para cada vértice destino há possibilidade atualizar a sua capacidade (remoção + inserção)

$$\mathbf{S(V, E)} = O\ (V+E+V+V) \approx \mathbf{O\ (E)}$$

Lista de vértices do grafo

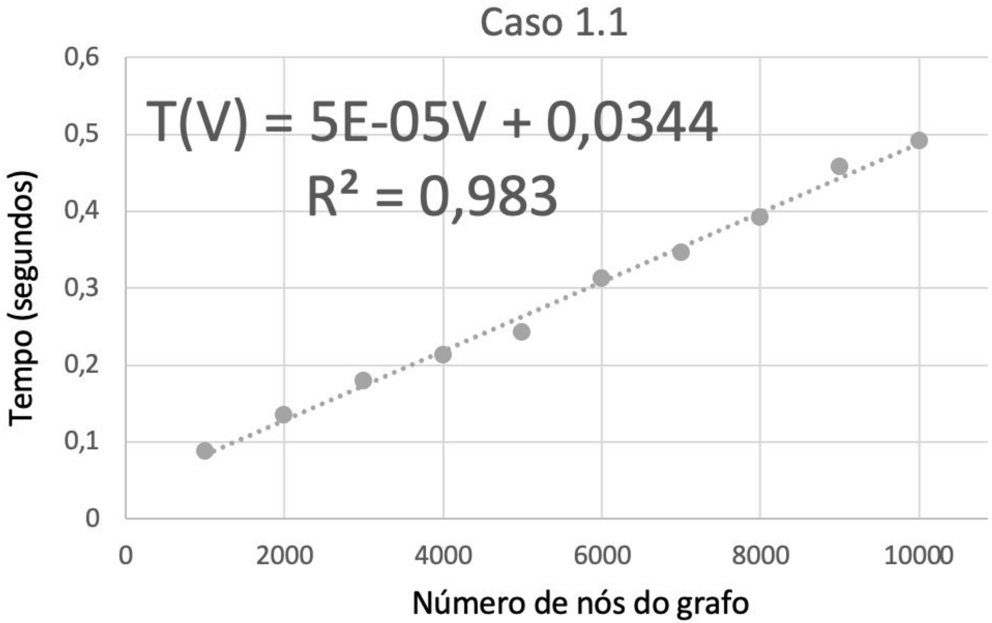
Lista de arestas do grafo

Set que guarda as capacidades ordenadas dos vértices

Lista que guarda o caminho encontrado

Cenário 1.1 - Avaliação empírica

Input V (número de nós)	**Tempo de execução (segundos)**
1000	0.088
2000	0.135
3000	0.180
4000	0.213
5000	0.213
6000	0.312
7000	0.346
8000	0.392
9000	0.458
10000	0.492



Variando o número de nós do grafo, a complexidade temporal do algoritmo é praticamente linear, o que condiz com a complexidade teórica apresentada no slide anterior

Cenário 1.2 - Formalização

Input:

- O - Nó de Origem do percurso desejado
- D - Nó de Destino do percurso desejado
- Grafo G(V, E) em que:
  - V - Número de vértices do Grafo G
  - E - Conjunto de arestas do Grafo G em que, para i $\in$ [0..N]:
    - E_i.origin - index do nó de origem da aresta
    - E_i.destiny - index do nó de destino da aresta
    - E_i.time - tempo, em horas, que a aresta demora a ser percorrida
    - E_i.capacity - capacidade, em quantidade de pessoas, da aresta

Output / Variáveis de decisão:

- C - número de máximo pessoas a viajar
- V1’ = {O, $V1'_1$, …, $V1'_{K-1}$, D} , lista ordenada de Vértices que formam o caminho de capacidade C.
- T - número de transbordos na viagem
- V2’ = {O, $V2'_1$, …, $V2'_{K-1}$, D} , lista ordenada de Vértices que formam o caminho de T transbordos.

Cenário 1.2 - Formalização

Restrições:

- $\forall e \in E, e_{origin} \in [1, N] \wedge e_{destiny} \in [1, N] \wedge e_{time} > 0 \wedge e_{capacity} \geq 0$
- $C = min(\{e.capacity | e \in E \wedge e.origin, e.destination \in V_1' \wedge V_1'i = e.origin \Rightarrow V_1'i + 1 = e.destination\})$
- $T = \#V_2 - 1$
- $C = min(\{e.capacity | e \in E \wedge e.origin, e.destination \in V_2' \wedge V_2'i = e.origin \Rightarrow V_2'i + 1 = e.destination\})$

Objetivo: $max(C) \wedge min(T)$

Cenário 1.2 - Algoritmos

Numa primeira fase, o cálculo do caminho com maior capacidade, repete-se o algoritmo de 1.1.

De seguida, após um novo reset a todos os nós do grafo, realiza-se uma pesquisa em largura, BFS, partindo do nó de origem. Assim garante-se que a distância à origem, medida em número de nós percorridos, é mínima partindo de qualquer outro nó. Partindo da premissa que o grafo é acíclico e que o número de transbordos é igual à ao número de nós visitados no percurso menos um, então a solução encontrada é ótima.

O algoritmo é auxiliado por uma fila e esta garante por um lado a visita em largura ao inserir cada nó antes dos seus descendentes e por outro operações em tempo constante, O(1).

```
int Graph::BFS(int origin, int destiny) {

    reset();
    int capacity = INF;
    queue<int> visitedNodes = {};
    visitedNodes.push(origin);
    nodes[origin].visited = true;

    while (!visitedNodes.empty()) {

        int node = visitedNodes.front();
        visitedNodes.pop();

        for (Edge e : nodes[node].adjacent) {
            int n = e.dest;
            if (!nodes[n].visited && e.capacity > 0) {
                visitedNodes.push(n);
                nodes[n].visited = true;
                nodes[n].capacity = e.capacity;
                nodes[n].parent = node;
                capacity = min(capacity, e.capacity);
            }
        }
    }
    return capacity;
}
```

Cenário 1.2 - Complexidade

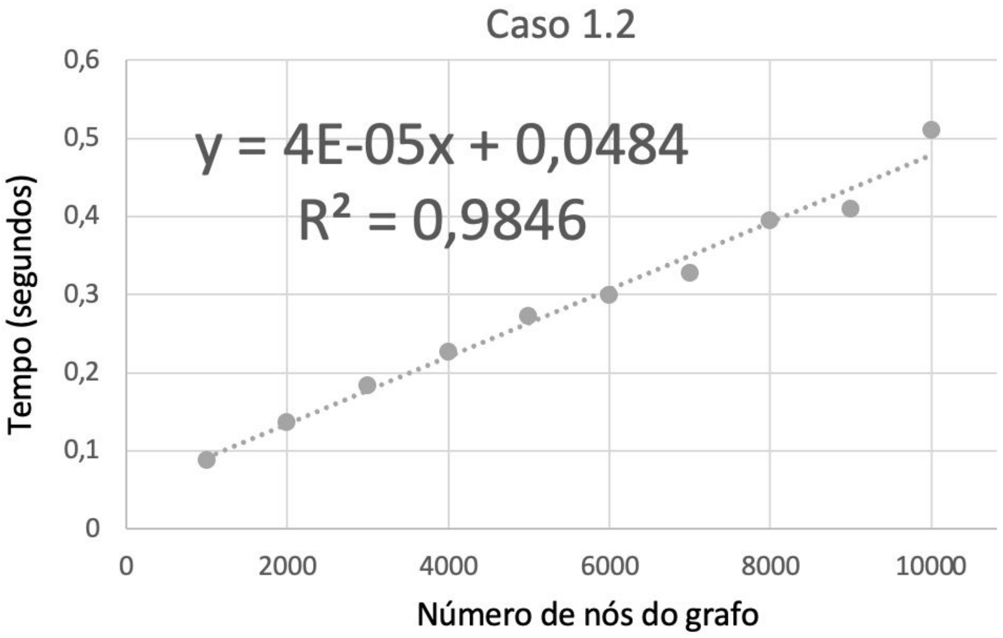
Seja V o número de Vértices e E o número de Arestas, onde E ≥ V:

$$\mathbf{T(V, E)} = O\ (\ V\ \underbrace{+\ V*\log(V) + V(\ \log(V) + E*\log(V))}_{\text{Caso 1.1}}\ +\ \underbrace{V}_{\text{Reset de todos os nós}} +\ \underbrace{V}_{\text{BFS para encontrar o caminho com o menor número de transbordos}} +\ E\ ) \approx \mathbf{O\ (\ V*E\ \log(V))}$$

$$\mathbf{S(V, E)} = O\ (\underbrace{V}_{\text{Lista de vértices do grafo}}+\underbrace{E}_{\text{Lista de arestas do grafo}}+\underbrace{V}_{\text{Set que guarda as capacidades ordenadas dos vértices}}+\underbrace{V}_{\text{Queue que guarda os vértices}}+\underbrace{V}_{\text{Lista que guarda o caminho encontrado}}) \approx \mathbf{O\ (E)}$$

Cenário 1.2 - Avaliação empírica

Input V (número de nós)	**Tempo de execução (segundos)**
1000	0,088
2000	0,137
3000	0,183
4000	0,227
5000	0,272
6000	0,299
7000	0,328
8000	0,395
9000	0,409
10000	0,51



Variando o número de nós do grafo, a complexidade temporal do algoritmo é praticamente linear, o que condiz com a complexidade teórica apresentada no slide anterior

Cenário 2.1 - Formalização

Input:

- C - Número de pessoas que compõe o grupo
- O - Nó de Origem do percurso desejado
- D - Nó de Destino do percurso desejado
- Grafo G(V, E) em que:
  - V - Número de vértices do Grafo G
  - E - Conjunto de arestas do Grafo G em que, para i ∈ [0..N]:
    - E_i.origin - index do nó de origem da aresta
    - E_i.destiny - index do nó de destino da aresta
    - E_i.time - tempo, em horas, que a aresta demora a ser percorrida
    - E_i.capacity - capacidade, em quantidade de pessoas, da aresta

Output / Variáveis de decisão:

- L = {V_0', V_1', … V_i'}, i ∈ [0..N], tal que:
  - V_i' = {O, V'_1, …, V'_{k-1}, D} , lista ordenada de Vértices que formam um caminho de O até D.

Restrições:

- $\forall e \in E, e_{origin} \in [1, N] \wedge e_{destiny} \in [1, N] \wedge e_{time} > 0 \wedge e_{capacity} \geq 0$
- $\forall V' \in L \quad \forall v \in V', v \in [1, N]$

Objetivo:

- $C \leq \sum_{k=0}^{N} (min(\{e.capacity | e \in E \quad \wedge e.origin, e.destination \in L_k \wedge L_K[i] = e.origin \Rightarrow L_k[i+1] = e.destination\})$

Cenário 2.1 - Algoritmos

Baseia-se em parte no Algoritmo de Edmonds-Karp.

Sempre que existam N pessoas a alocar, N > 0, uma BFS determina um caminho entre o nó de origem e nó de destino e retorna a capacidade, C, desse caminho. Todas as capacidades das arestas são decrementadas em C unidades e, para a próxima iteração, ficam a faltar N - C pessoas.

Se durante a iteração não houver caminho possível (ou seja, o caminho acaba num vértice V, tal que V.parent = -1 mas V não é a origem) e ainda existirem pessoas a alocar, então é exibida uma mensagem de erro. Caso contrário, no fim, é mostrado cada caminho assim como a correspondente capacidade máxima.

```
void Graph::case2_a(int origin, int destiny, int groupSize) {

    vector<vector<int>> pathList;
    int remainderSize = groupSize;
    vector<int> capacities;
    vector<int> path;

    while (remainderSize>0){

        path.clear();
        int capacity = BFS(origin, destiny);
        if(pathBuild(origin, destiny, path, capacity)) {
            cerr << "Error: There is no path between node " << origin << " and node " << destiny
            return;
        }
        remainderSize = verifyFoundPath(pathList, path, capacities, capacity, remainderSize);
    }
    showPathCase2(pathList, capacities);
}
```

```
int Graph::pathBuild(int origin, int destiny, vector<int> &path, int capacity){

    int currentNode = destiny;
    while (nodes[currentNode].parent != -1) {
        path.push_back(currentNode);
        for (Edge &e : nodes[nodes[currentNode].parent].adjacent){
            if (e.dest == currentNode) {
                e.capacity -= capacity;
            }
        }
        currentNode = nodes[currentNode].parent;
    }
    if (currentNode != origin) {
        return 1;
    } else path.push_back(currentNode);
    reverse(path.begin(), path.end());
    return 0;
}
```

Cenário 2.1 - Complexidade

Seja C a dimensão do grupo pretendida, V o número de Vértices e E o número de Arestas, onde E ≥ V:

$$\mathbf{T(C, V, E)} = O\ (\ C*(V + \underline{V + E} + V + E) + C*V) \approx \mathbf{O\ (C*E)}$$

Na pior das hipóteses, só há caminhos de capacidade 1, pelo que existirão C iterações

Reset de todos os nós

BFS para determinação do próximo caminho

Criação do novo caminho

Atualização das capacidades das arestas visitadas

Imprimir resultados. No máximo existem C caminhos.

$$\mathbf{S(C, V, E)} = O\ (\ V + E + V + C*V\ ) \approx \mathbf{O\ (\ C*V\ )}$$

Lista de vértices do grafo

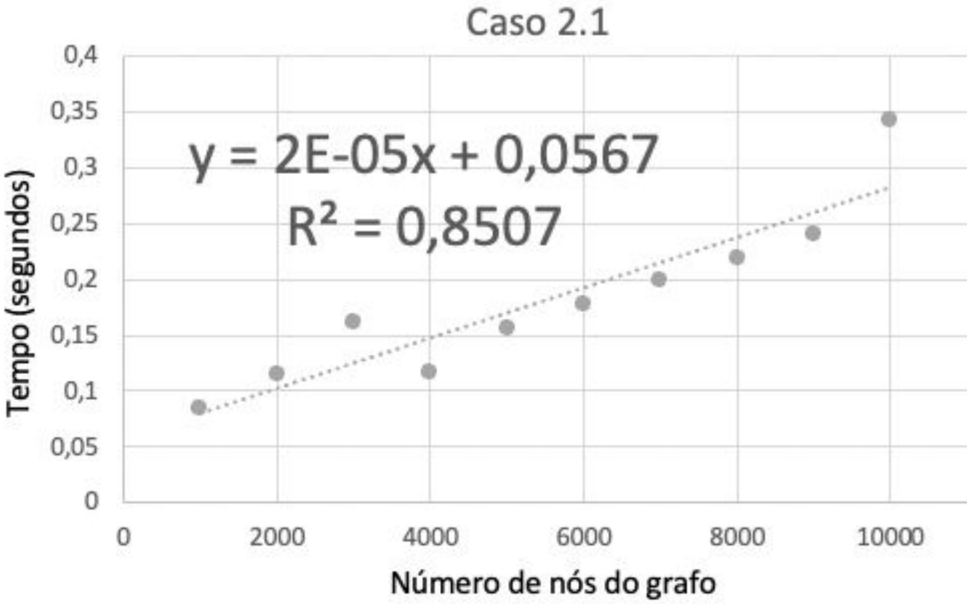
Lista de arestas do grafo

Cada novo caminho gerado

Os resultados são guardados numa matriz com, no máximo, C linhas

Cenário 2.1 - Avaliação empírica

Input V (número de nós)	**Tempo de execução (segundos)**
1000	0,085
2000	0,116
3000	0,162
4000	0,117
5000	0,156
6000	0,177
7000	0,199
8000	0,219
9000	0,241
10000	0,342



Variando o número de nós do grafo, a complexidade temporal do algoritmo é praticamente linear, o que condiz com a complexidade teórica apresentada no slide anterior

Cenário 2.2 - Formalização

Input:

- C - Número de pessoas que compõe o grupo anterior
- L - Lista de caminhos utilizados pelo grupo anterior
- C’ - Número de pessoas que compõe o novo grupo
- O - Nó de Origem do percurso desejado
- D - Nó de Destino do percurso desejado
- Grafo G(V, E) em que:
  - V - Número de vértices do Grafo G
  - E - Conjunto de arestas do Grafo G em que, para i ∈ [0..N]:
    - E_i.origin - index do nó de origem da aresta
    - E_i.destiny - index do nó de destino da aresta
    - E_i.time - tempo, em horas, que a aresta demora a ser percorrida
    - E_i.capacity - capacidade, em quantidade de pessoas, da aresta

Output / Variáveis de decisão:

- L’ = {V_0’, V_1’, … V_i’}, i ∈ [0..N], tal que:
  - V_i’ = {O, V’$_1$, …, V’$_{k-1}$, D} , lista ordenada de Vértices que formam um caminho de O até D.

Restrições:

- $\forall e \in E, e_{origin} \in [1, N] \wedge e_{destiny} \in [1, N] \wedge e_{time} > 0 \wedge e_{capacity} \geq 0$
- $\forall V' \in L \quad \forall v \in V', v \in [1, N]$

Objetivo:

- $C \leq \sum_{k=0}^{N}(min(\{e.capacity | e \in E \quad \wedge e.origin, e.destination \in L_k \wedge L_K[i] = e.origin \Rightarrow L_k[i+1] = e.destination\})$

Cenário 2.2 - Algoritmos

Baseia-se no mesmo algoritmo usado no cenário 2.1, porém com adaptações.

Diferente do cenário anterior, o algoritmo recebe uma lista de caminhos e a dimensão do grupo para qual a lista foi planejada.

Caso a nova dimensão do grupo desejada seja menor ou igual ao input recebido, os caminhos originais serão retornados.

No começo do algoritmo, a capacidade de cada caminho recebido é preenchida. Para cada encaminhamento, calcula-se a capacidade de fluxo máximo. Depois, atualiza-se o fluxo do caminho sendo o fluxo máximo do caminho ou o fluxo restante pretendido (o mínimo dos valores).

O algoritmo também verifica se o caminho já existe. Caso isso se verifique, a quantidade de pessoas que utilizam o caminho é acrescida, evitando a existência de caminhos duplicados no resultado.

```
void Graph::case2_b(int origin, int destiny, vector<vector<int>> pathList, int oldGroupSize, int newGroupSize) {

    int remainderSize = newGroupSize - oldGroupSize;
    vector<int> capacities;
    vector<int> path;

    for(vector<int> curPath: pathList){

        int pathCapacity = INF;
        for(int i=0; i<curPath.size()-1; i++){
            for(auto j: nodes[curPath[i]].adjacent){
                if(j.dest == curPath[i+1]){
                    pathCapacity = min(pathCapacity, j.capacity);
                }
            }
        }

        for(int i=0; i<curPath.size()-1; i++){
            for(auto &j: nodes[curPath[i]].adjacent){
                if(j.dest == curPath[i+1]){
                    j.capacity -= min(pathCapacity, oldGroupSize);
                }
            }
        }
        capacities.push_back(min(pathCapacity, oldGroupSize));
        oldGroupSize -= pathCapacity;
    }

    if(newGroupSize > oldGroupSize){
        while (remainderSize>0){

            path.clear();
            int capacity = BFS(origin, destiny);
            if(pathBuild(origin, destiny, path, capacity)) {
                cerr << "Error: There is no path between node " << origin << " and node " << destiny << " with desired c
                return;
            }
            remainderSize = verifyFoundPath(pathList, path, capacities, capacity, remainderSize);
        }
    }
    showPathCase2(pathList, capacities);
}
```

Cenário 2.2 - Complexidade

Seja C a dimensão do grupo pretendida, L o número de caminhos dado, V o número de Vértices e E o número de Arestas, onde $E \geq V$:

$$\mathbf{T(L, C, V, E)} = O\left(L*(V+E) + \underline{C*(V + V + E + V + E)} + C*V \right) \approx \mathbf{O\ (C*E)}$$

Atualização das capacidades das arestas do grafo original de acordo com cada caminho recebido

Abordagem semelhante ao problema 2.1, com BFS para ajustar o encaminhamento. Na pior das hipóteses, se existirem apenas caminhos de capacidade 1, o ciclo repete-se C vezes.

Mostrar resultados

$$\mathbf{S(C, V, E)} = O\ (V + E + V + C*V) \approx \mathbf{O\ (C*V)}$$

Lista de vértices do grafo

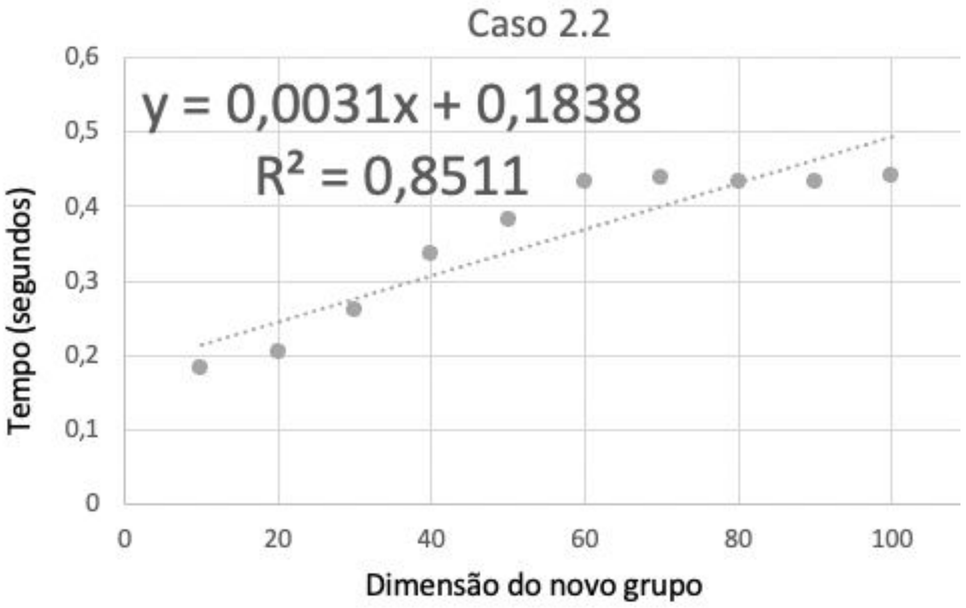
Lista de arestas do grafo

Cada novo caminho gerado

Os resultados são guardados numa matriz com, no máximo, C linhas

Cenário 2.2 - Avaliação empírica

Input C (dimensão do grupo)	**Tempo de execução (segundos)**
10	0,184
20	0,206
30	0,262
40	0,337
50	0,38
60	0,433
70	0,435
80	0,434
90	0,433
100	0,443



Variando a dimensão do grupo, a complexidade temporal do algoritmo é praticamente linear, o que condiz com a complexidade teórica apresentada no slide anterior

Cenário 2.3 - Formalização

Input:

- O - Nó de Origem do percurso desejado
- D - Nó de Destino do percurso desejado
- Grafo G(V, E) em que:
  - V - Número de vértices do Grafo G
  - E - Conjunto de arestas do Grafo G em que, para i ∈ [0..N]:
    - E_i.origin - index do nó de origem da aresta
    - E_i.destiny - index do nó de destino da aresta
    - E_i.time - tempo, em horas, que a aresta demora a ser percorrida
    - E_i.capacity - capacidade, em quantidade de pessoas, da aresta

Output. Variáveis de decisão:

- C - Número máximo de pessoas a viajar no grafo G
- L’ = {V_0’, V_1’, … V_i’}, i ∈ [0..N], tal que:
  - V_i’ = {O, $V’_1$, …, $V’_{k-1}$, D} , lista ordenada de Vértices que formam um caminho de O até D.

Restrições:

- $\forall e \in E, eorigin \in [1, N] \wedge edestiny \in [1, N] \wedge etime > 0 \wedge ecapacity \geq 0$
- $\forall V' \in L \quad \forall v \in V', v \in [1, N]$
- $C = min(\{e.capacity | e \in E \wedge e.origin, e.destination \in L_k \wedge L_K[i] = e.origin \Rightarrow L'_k[i+1] = e.destination\})$

Objetivo: $max(C)$

Cenário 2.3 - Algoritmos

Da mesma forma que o algoritmo anterior, o cenário 2.3 utiliza o algoritmo do cenário 2.1 com pequenos ajustes.

O algoritmo é cíclico e possui término quando não é possível encontrar um novo encaminhamento, diferente do cenário 1, que termina quando os caminhos encontrados já são suficientes para a dimensão do grupo.

```cpp
void Graph::case2_c(int origin, int destiny) {

    vector<vector<int>> pathList;
    vector<int> capacities;
    vector<int> path;

    while (true){

        path.clear();
        int capacity = BFS(origin, destiny);
        if(pathBuild(origin, destiny, path, capacity)) {
            break;
        }
        verifyFoundPath(pathList, path, capacities, capacity);
    }

    int totalCapacity = showPathCase2(pathList, capacities);
    cout << "Total capacity: " << totalCapacity << endl;
}
```

Cenário 2.3 - Complexidade

Seja V o número de Vértices e E o número de Arestas, onde E ≥ V:

$$\mathbf{T(V, E)} = O\,(N^*(V + E + V^*E)) \approx \mathbf{O\,(N^*V^*E)}$$

Número de caminhos a encontrar, ou seja, o número de iterações

Abordagem semelhante ao problema 2.1, com BFS para ajustar o encaminhamento.

$$\mathbf{S(V, E)} = O\,(\,V + E + V + C^*V\,) \approx \mathbf{O\,(\,C^*V\,)}$$

Lista de vértices do grafo

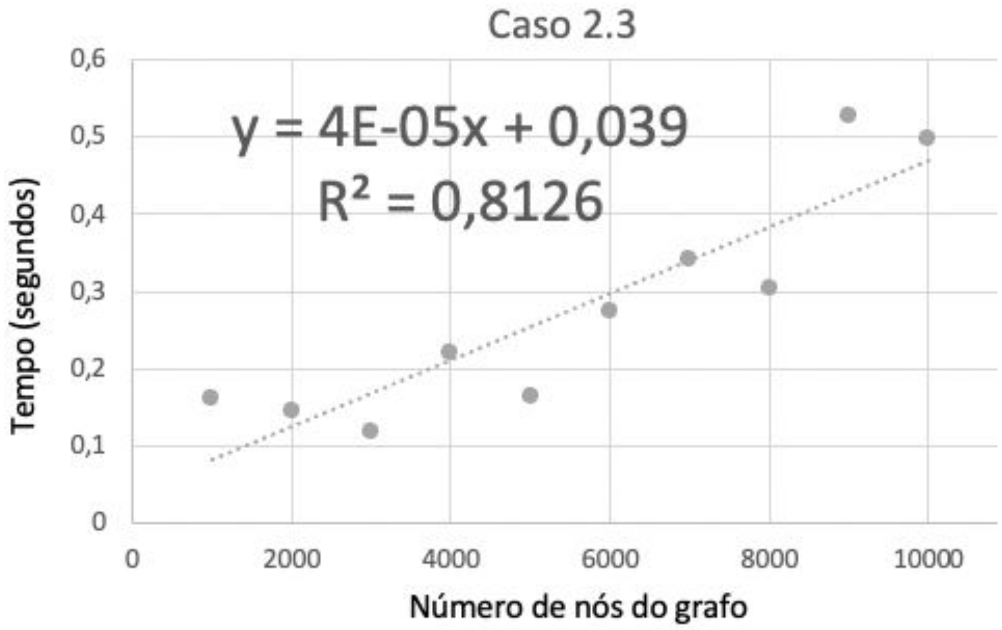
Lista de arestas do grafo

Cada novo caminho gerado

Os resultados são guardados numa matriz com, no máximo, C linhas

Cenário 2.3 - Avaliação empírica

Input V (número de nós)	Tempo de execução (segundos)
1000	0,162
2000	0,146
3000	0,118
4000	0,221
5000	0,165
6000	0,275
7000	0,343
8000	0,305
9000	0,529
10000	0,498



Variando o número de nós, a complexidade temporal do algoritmo é praticamente linear, o que condiz com a complexidade teórica apresentada no slide anterior

Cenário 2.4 - Formalização

Input:

- C - Número de pessoas que compõe o grupo
- O - Nó de Origem do percurso desejado
- D - Nó de Destino do percurso desejado
- L = {V_0, V_1, … V_i}, i ∈ [0..N], tal que:
  - V_i = {O, V_1, …, V_{k-1}, D} , um subconjunto de Vértices que formam um caminho de O até D.
- Grafo G(V, E) em que:
  - V - Número de vértices do Grafo G
  - E - Conjunto de arestas do Grafo G em que, para i ∈ [0..N]:
    - E_i.origin - index do nó de origem da aresta
    - E_i.destiny - index do nó de destino da aresta
    - E_i.time - tempo, em horas, que a aresta demora a ser percorrida
    - E_i.capacity - capacidade, em quantidade de pessoas, da aresta

Output. Variáveis de decisão:

- T - Tempo mínimo para o grupo se reunir no nó de origem

Restrições:

- $O \in V$
- $D \in V$
- $\forall V' \in L \quad \forall v \in V', v \in V$
- $T > 0$
- $T = max(\quad \sum_{k=0}^{N} \{e.time | e.origin \in E \wedge e.destination \in E \wedge origin \in L_k \wedge destination \in L_k \wedge L_k[i] = origin \Rightarrow L_k[i+1] = destination\})$

Objetivo: $min(T)$

Cenário 2.4 - Algoritmos

Começa-se por definir o tempo máximo de chegada ao destino igual a zero (endTime = 0).

O algoritmo percorre a lista de caminhos, e para cada encaminhamento, calcula o tempo necessário para completá-lo. Para isso, os tempos das arestas do caminho são acumulados. Depois de efetuar o cálculo do caminho, caso o tempo seja maior que o endTime guardado, o valor é atualizado.

```cpp
void Graph::case2_d(int origin, int destiny, vector<vector<int>> pathList){

    if(pathList.empty()) return;
    int endTime = 0;

    for(vector<int> curPath: pathList){

        int groupTime = 0;
        for(int i=0; i<curPath.size()-1; i++){
            for(auto j: nodes[curPath[i]].adjacent){
                if(j.dest == curPath[i+1]){
                    groupTime += j.duration;
                }
            }
        }
        endTime = max(endTime,groupTime);
    }
    cout << "Minimum amount of time for the arrival of the whole group: " << endTime << " hours" << endl;
}
```

Cenário 2.4 - Complexidade

Seja L o número de caminhos dado, V o número de Vértices e E o número de Arestas, onde E ≥ V:

$$\mathbf{T(L, V, E) = O\ (L*V*E)}$$

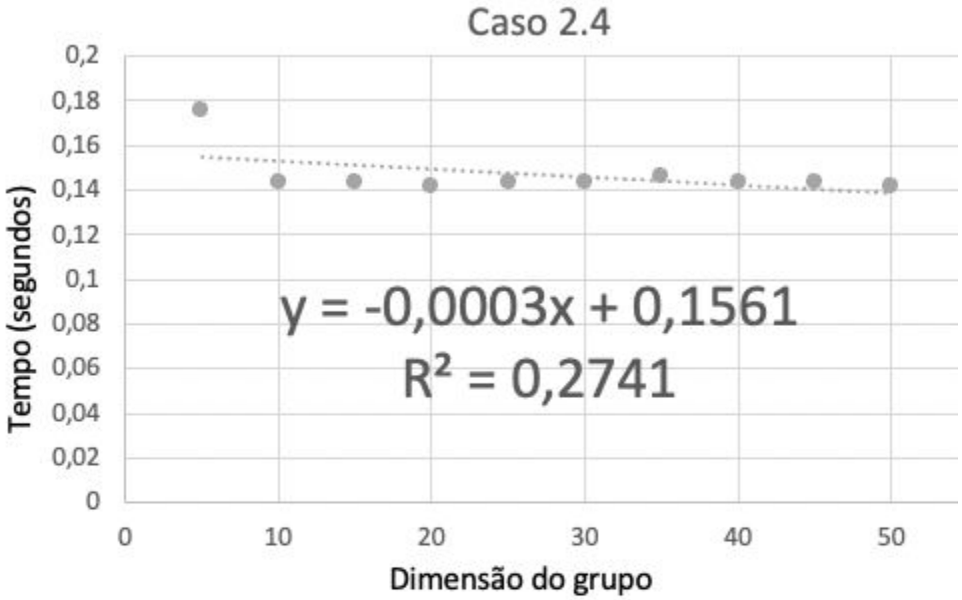
Para cada caminho dado e para cada par de nós contínuos nesse caminho, consultar o tempo a percorrer a aresta que os separa

$$\mathbf{S(L, V, E)} = O\ (E)$$

Foi considerado que o número de arestas do grafo original é muito superior a qualquer outro input

Cenário 2.4 - Avaliação empírica

Input C (dimensão do grupo)	Tempo de execução (segundos)
5	0,175
10	0,143
15	0,144
20	0,142
25	0,144
30	0,144
35	0,145
40	0,143
45	0,144
50	0,142



Variando o número de caminhos dados por input (e consequentemente o número de pessoas do grupo inicial), a complexidade temporal do algoritmo é praticamente constante para o mesmo grafo. De facto, os vértices e as arestas têm um maior peso na complexidade.

Cenário 2.5 - Formalização

Input:

- C - Número de pessoas que compõe o grupo
- O - Nó de Origem do percurso desejado
- D - Nó de Destino do percurso desejado
- L = $\{V_0, V_1, \ldots V_i\}$, i ∈ [0..N], tal que:
  - $V_i = \{O, V_1, \ldots, V_{k-1}, D\}$, um subconjunto de Vértices que formam um caminho de O até D.
- Grafo G(V, E) em que:
  - V - Número de vértices do Grafo G
  - E - Conjunto de arestas do Grafo G em que, para i ∈ [0..N]:
    - E_i.origin - index do nó de origem da aresta
    - E_i.destiny - index do nó de destino da aresta
    - E_i.time - tempo, em horas, que a aresta demora a ser percorrida
    - E_i.capacity - capacidade, em quantidade de pessoas, da aresta

Output/ Variáveis de decisão:

- V' = $\{V'_1 .. V'_K\}$ - Lista ordenada de vértices de G em que há um grupo de C pessoas que esperam por outra parte do grupo
- T' = $\{T'_1 .. T'_K\}$ - Tempos de espera, em horas, para cada vértice de V'

Restrições:

- $O \in V$
- $D \in V$
- $\forall V' \in L \quad \forall v \in V', v \in V$
- $\forall i \in [0..N] : V'_i \in V \land T'_i > 0$

Objetivo: $min(T_i), i \in [1..N]$

Cenário 2.5 - Algoritmos

O algoritmo começa por marcar todos os nós que fazem parte de pelo menos um encaminhamento da lista recebida.

Em segundo lugar, para cada nó marcado é calculado o earliestStart, utilizado-se o Método do Caminho Crítico, implementado com uma pilha.

Após, é calculado o “earliestArrival” de cada nó. Isso é feito percorrendo as arestas de cada nó (A), e para o nó destino da aresta (B) seleciona-se o mínimo entre o “earliestArrival” guardado em B e o “earliestArrival de A + tempo para percorrer aresta”.

Por fim, para cada nó que possui mais do que um grupo a “visitar”, calcula-se, o tempo máximo de espera sendo o earliestStart - earliestArrival.

```cpp
void Graph::case2_e(int origin, int destiny, vector<vector<int>> pathList){

    reset();

    for (vector<int> path : pathList) {
        for (int i = 0 ; i < path.size() -1 ; i++) {
            for (Edge &e : nodes[path[i]].adjacent) {
                if (e.dest == path[i+1]) {
                    e.visit = true;
                    nodes[e.dest].visited = true;
                    nodes[e.dest].degreeE++;
                }
                nodes[path[i]].visited = true;
            }
        }
    }

    getEarliestStart();
    getEarliestArrival();

    for (int i = 1 ; i < nodes.size() ; i++) {
        int waiting = nodes[i].earliestStart - nodes[i].earliestArrival;
        if (nodes[i].visited && waiting > 0)
            cout << "Node: " << i << " can have a waiting time of " << waiting << " hours" << endl;
    }
}
```

```cpp
void Graph::getEarliestStart(){
    stack<int> S;
    for(int i = 1 ; i < nodes.size() ; i++){
        if(nodes[i].visited && nodes[i].degreeE==0){
            S.push(i);
        }
    }
    int minDuration = -1;
    int vFinal;

    while(!S.empty()){
        int v = S.top();
        S.pop();

        if(minDuration < nodes[v].earliestStart){
            minDuration = nodes[v].earliestStart;
            vFinal = v;
        }

        for(auto &w: nodes[v].adjacent){
            if( nodes[w.dest].earliestStart < nodes[v].earliestStart + w.duration){
                nodes[w.dest].earliestStart = nodes[v].earliestStart + w.duration;
            }
            nodes[w.dest].degreeE--;
            if(nodes[w.dest].degreeE==0){
                S.push(w.dest);
            }
        }
    }
}
```

```cpp
void Graph::getEarliestArrival(){
    for (int i = 1 ; i < nodes.size() ; i++) {
        if (nodes[i].visited) {
            for (Edge e : nodes[i].adjacent) {
                if (nodes[e.dest].visited && e.visit) {
                    nodes[e.dest].earliestArrival = min(nodes[e.dest].earliestArrival, e.duration + nodes[i].earliestStart);
                }
            }
        }
    }
}
```

Cenário 2.5 - Complexidade

Seja C a dimensão do grupo pretendida, L o número de caminhos dado, V o número de Vértices e E o número de Arestas, onde E ≥ V:

$$\mathbf{T(L, V, E)} = O\ (\ V + (L*V*E) + (V+V*E) + (V*E) + V\ ) \approx \mathbf{O\ (L*V*E)}$$

Reset dos atributos dos vértices

Marcar os nós a visitar e calcular o grau dos mesmos

Calcular o earliestStart dos nós
[percorrer os nós e fazer push na stack -> percorrer a stack e as arestas dos nós]

Calcular o earliestArrival dos nós
[percorrer os nós, e arestas de cada nó]

Percorrer os nós e devolver o fazer cout quando necessário

$$\mathbf{S(L, V, E)} = O\ (\ L*V + V + E + V) \approx \mathbf{O\ (\ L*V\ )}$$

Lista de caminhos (majorando a quantidade de nós de cada caminho)

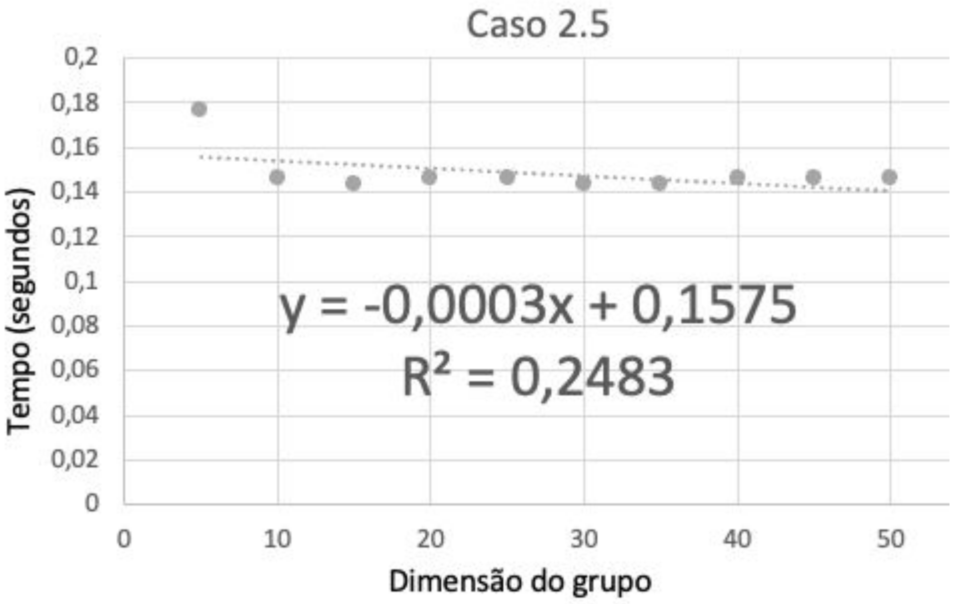
Lista de vértices do grafo

Lista de arestas do grafo

Stack usada no algoritmo para calcular o earliestStart

Cenário 2.5 - Avaliação empírica

Input C (dimensão do grupo)	Tempo de execução (segundos)
5	0,177
10	0,146
15	0,144
20	0,145
25	0,146
30	0,144
35	0,144
40	0,146
45	0,146
50	0,146



Variando o número de caminhos dados por input (e consequentemente o número de pessoas do grupo inicial), a complexidade temporal do algoritmo é praticamente constante para o mesmo grafo. De facto, os vértices e as arestas têm um maior peso na complexidade.

Solução Algorítmica a destacar

No cenário 2.2, o input dos caminhos dá-se através de um ficheiro com a dimensão do grupo na primeira linha e encaminhamentos nas linhas restantes. O algoritmo é capaz de calcular uma possível divisão do grupo nos diferentes encaminhamentos, e após isso continua o algoritmo do 2.1 com essa informação.

No cenário 2.5, para cada nó de interesse (e apenas nós de interesse) calculamos o “earliestStart” e o “earliestArrival” o que facilita no cálculo do tempo máximo de espera em cada nó, sendo earliestStart - earliestArrival.

Dificuldades encontradas

- Implementar o Algoritmo de Edmonds-Karp adaptado às nossas necessidades.

Autoavaliação

- Fábio Araújo de Sá, **50%**
- Marcos William Ferreira Pinto, **50%**