Agência de viagens - Transporte de grupos de pessoas

Desenho de Algoritmos - Grupo 113

Gustavo Costa - up202004187
José Araújo - up202007921
Ricardo Cavalheiro - up202005103

Descrição do Problema Geral

O problema geral trata-se de um problema de gestão e otimização na área do transporte de grupos de pessoas.

Para este trabalho, 2 cenários foram criados, com o objetivo de resolver de 2 sub-problemas:

-Cenário 1: transporte de grupos indivisíveis,

-Cenário 2: transporte de grupos divisíveis.

Descrição

- O Cenário 1 é relativo a transportes de grupos que não se separam, subdividindo-se em dois problemas:

1. Maximização da dimensão do grupo e indicar um qualquer encaminhamento.

2. Maximização da dimensão do grupo, minimizando o número de transbordos, sem, no entanto, privilegiar um dos critérios relativamente ao outro, apresentando as alternativas que sejam pareto-ótimas, se existirem. Tal significa que um grupo maior pode ser transportado se se admitir mais transbordos.

Formalização

Cenário 1.1 - maximização da dimensão do grupo

- Input:
  - dataset com informação sobre os nós (N) e arestas (E)
- Output:
  - número máximo de passageiros na viagem (C)
  - número de transbordos (T)
  - respetivo percurso (P)
- Variáveis de decisão:
  - C : capacidade máxima de passageiros
- Funções-objetivo:
  - maximizar C (capacidade de passageiros)
- Restrições e Domínios de valores:
  - $0 \leq T$, $0 \leq C$, para todo o $T, C \in \mathbb{N}$ (se não for encontrado um caminho ambos são 0)
  - $1 \leq N$, $0 \leq E$ para todo o $N, E \in \mathbb{N}$
- Objetivos:
  - maximizar a dimensão do grupo na viagem

Algoritmos Relevantes

Cenário 1.1 - maximização da dimensão do grupo

- O algoritmo que utilizamos para solucionar este problema foi uma adaptação do Algoritmo de Dijkstra, que tendo em conta um nó de entrada e de saída descobre o percurso que nos permite transportar o grupo com maior dimensão utilizando uma fila de prioridade (*MaxHeap*). Este algoritmo não tem em conta o número de transbordos.

- Após obtermos o percurso com a maior dimensão apresentamos ao utilizador da seguinte forma:

```
int Ctcp::getMaxCapacityPath(int st, int end, list<int> *path){
    for (auto& v:nodes) {
        v.visited = false;
        v.distance = 0;
        v.parent = -1;
    }

    nodes[0].visited = true;
    nodes[st].distance = INT_MAX;
    MaxHeap<int,double> q(n, notFound: -1);

    for(int i = 1; i <= n ; i++){
        q.insert(i, value: nodes[i].distance);
    }
    while(q.getSize() > 0){
        int u = q.removeMax();
        for(const auto& v: nodes[u].adj){
            if(nodes[v.dest].distance < min(nodes[u].distance, v.capacity)){
                nodes[v.dest].distance = min(nodes[u].distance, v.capacity);
                nodes[v.dest].parent = u;
                q.increaseKey(v.dest,v.capacity);
            }
        }
    }
```

```
Maximum number of passengers: 16
Transbordos: 13
PATH: 1 -> 239 -> 287 -> 125 -> 12 -> 148 -> 130 -> 71 -> 249 -> 252 -> 109 -> 255 -> 92 -> 300
```

Ex: Entrada - nó 1; Saída - nó 300

Input: in04_b.txt

```
We couldn't find a path for you!
```

Caso não haja um percurso entre os dois locais o utilizador é informado.

Análise de complexidade

Cenário 1.1 - maximização da dimensão do grupo

Complexidade temporal:

- O algoritmo utilizado apresenta complexidade temporal O((|V| + |E|) log2 |V|).

Complexidade espacial:

- A complexidade espacial associada é S(V,E) = O(V).

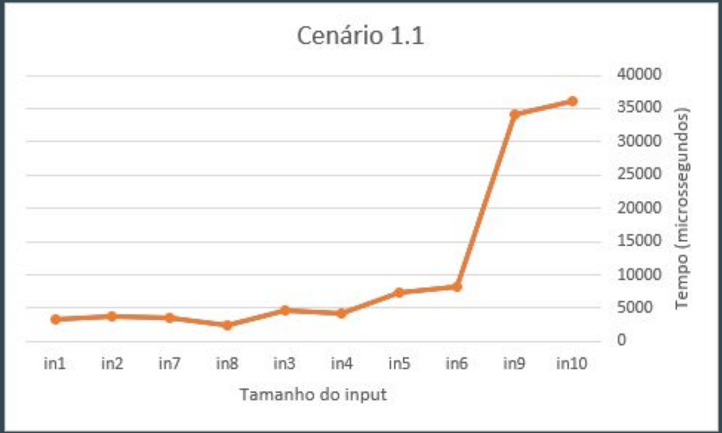
  É utilizado um vetor para armazenar o percurso obtido pelo algoritmo e foi criado uma MaxHeap para ser possível executar o algoritmo.

Nota: V - número de nós (locais)

E - número de ramos (veículos)

Resultados da Avaliação Empírica

Cenário 1.1 - maximização da dimensão do grupo



$$V = N$$

$$E = N(N-1)$$

$$(|V| + |E|) \log 2\ |V| = (N + N(N-1)) \text{ Log2 } N$$

Formalização

Cenário 1.2 - maximizar dimensão do grupo e minimizar transbordos

- Input:
  - dataset com informação sobre os nós (N) e arestas (E)
- Output:
  - número máximo de passageiros na viagem (C)
  - número de transbordos (T)
  - respetivo percurso (P)
- Variáveis de decisão:
  - C : capacidade máxima de passageiros
  - T : nº de transbordos no percurso
- Funções-objetivo:
  - maximizar C e minimizar T (sem privilegiar um dos critérios)
- Restrições e Domínios de valores:
  - $0 \leq T, 0 \leq C$, para todo o $T, C \in \mathbb{N}$ (se não for encontrado um caminho ambos são 0)
  - $2 \leq N, 1 \leq E$ para todo o $N, E \in \mathbb{N}$
- Objetivos:
  - maximizar a dimensão do grupo na viagem
  - minimizar o nº de transbordos

Algoritmos relevantes

Cenário 1.2 - maximizar dimensão do grupo e minimizar transbordos

- Este subproblema por sua vez é mais trabalhoso porque é necessário encontrar os percursos que maximizem a dimensão do grupo minimizando os transbordos, sem privilegiar um dos critérios.

- Começamos por aplicar o algoritmo ***Breadth First Search*** para encontrarmos o caminho mais curto (com menos transbordos). Foi adicionado um atributo booleano aos nós do grafo que é inicializado a falso menos aos nós pertencentes ao caminho mais curto.

- Após isso, é feito um ciclo que acaba apenas quando todos os nós foram utilizados (isto é, o atributo seja verdadeiro para todos os nós). Dentro desse ciclo é adicionado de cada vez um novo nó para ser utilizado e é chamado o algoritmo do cenário 1.1 adaptado para usar apenas os nós que sejam verdadeiros, encontrando assim todos os percursos que maximizem a dimensão do grupo tendo em conta os transbordos.

```
void Ctcp::bfs2(int v, int end) {
    for (int i=1; i<=n; i++) nodes[i].visited = false;
    std::queue<int> q; // queue of unvisited nodes
    q.push(v);
    nodes[v]. visited = true;
    while (!q.empty()) { // while there are still unvisited nodes
        int u = q.front(); q.pop();
        for (auto e : nodes[u].adj) {
            int w = e.dest;
            if(w == end) { nodes[w].parent = u; return;}
            if (!nodes[w].visited) {
                q.push(w);
                nodes[w].parent = u;
                nodes[w].visited = true;
            }
        }
    }
}
```

```
while(nodesUnused()){
    maxCapacity = getMaxCapacityWithUsableNodes(st,end, &pathAux2);
    if(solutions.back() != pathAux2){
        solutions.push_back(pathAux2);
        capacities.push_back(maxCapacity);
    }
    for(int i = 1; i <= n; i++){
        if(!nodes[i].use){
            nodes[i].use = true;
            break;
        }
    }
}
```

Algoritmos relevantes

Cenário 1.2 - maximizar dimensão do grupo e minimizar transbordos

- Por fim, é necessário filtrar os resultados para obter apenas aqueles que sejam realmente soluções pareto-ótimas.

- Esta filtragem é feita percorrendo a lista de percursos obtidos e removendo os caminhos em que exista uma solução melhor, isto é, com menos transbordos e uma capacidade maior ou igual, ou caso os transbordos sejam iguais remover aqueles que têm uma capacidade inferior.

```
for(int b = 0 ; b < s.size() ; b++){
    int capacity = capacities[b];
    unsigned long long transbordosAux = transbordos[b];
    for(int g = 0 ; g < s.size() ; g++){
        if(g == b) continue;
        if(transbordos[g]<transbordosAux && capacities[g]>= capacity){
            s.erase(s.begin()+b);
            capacities.erase(capacities.begin()+b);
            transbordos.erase(transbordos.begin()+b);
            b--;
            break;
        }

        if(transbordos[g]==transbordosAux && capacities[g]> capacity){
            s.erase(s.begin()+b);
            capacities.erase(capacities.begin()+b);
            transbordos.erase(transbordos.begin()+b);
            b--;
            break;
        }
        if(transbordos[g]>transbordosAux && capacities[g]<= capacity){
            s.erase(s.begin()+g);
            capacities.erase(capacities.begin()+g);
            transbordos.erase(transbordos.begin()+g);
            g--;
        }
    }
}
```

```
Solution 1:
Maximum number of passengers: 2
Transbordos: 4
PATH: 1 -> 6 -> 2 -> 49 -> 50

Solution 2:
Maximum number of passengers: 8
Transbordos: 5
PATH: 1 -> 30 -> 25 -> 12 -> 5 -> 50

Solution 3:
Maximum number of passengers: 10
Transbordos: 8
PATH: 1 -> 8 -> 42 -> 29 -> 44 -> 34 -> 12 -> 5 -> 50
```

Ex: Entrada - nó 1; Saída - nó 50

Input: in02_b.txt

Análise de complexidade

Cenário 1.2 - maximizar dimensão do grupo e minimizar transbordos

Complexidade temporal:

- O algoritmo utilizado apresenta complexidade temporal $O((|V| + |E|) \log_2 |V| * |V|)$.

Complexidade espacial:

- A complexidade espacial associada é $S(V,E) = O(\sum(i = 1,V-2) (A(i,V-2)) * V)$.

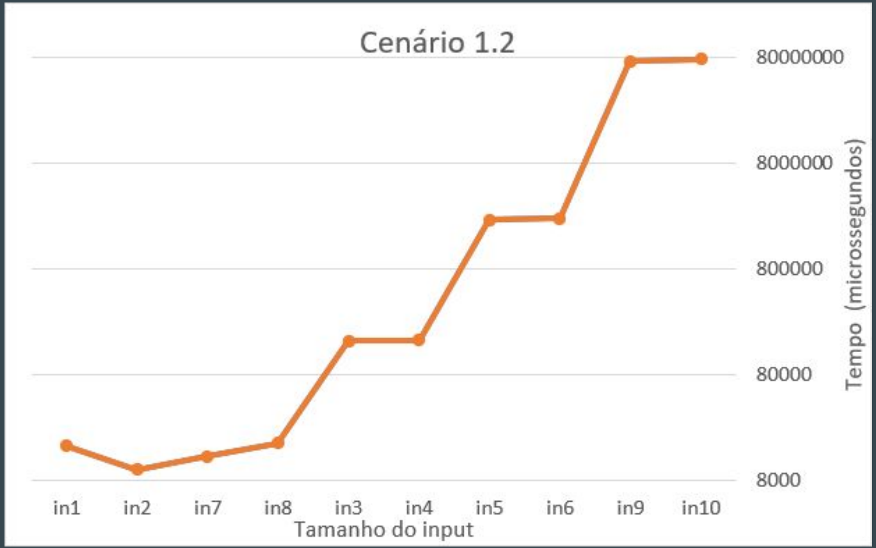
  É utilizada uma lista para armazenar todos os possíveis percursos que sejam soluções do problema.

Nota: V - número de nós (locais)

E - número de ramos (veículos)

Resultados da Avaliação Empírica

Cenário 1.2 - maximizar dimensão do grupo e minimizar transbordos



Descrição

- O Cenário 2 é relativo a transportes de grupos que se podem separar, subdividindo-se em 4 problemas:

1. Determinar um encaminhamento para um grupo, dada a sua dimensão, podendo esta aumentar de um número de unidades dado (2.1 e 2.2).
2. Determinar a dimensão máxima de um grupo e um encaminhamento (2.3).
3. Partindo de um encaminhamento, determinar quando é que o grupo se reuniria novamente no destino, no mínimo (2.4).
4. Admitindo que os elementos que saem de um mesmo local partem desse local à mesma hora (e o mais cedo possível), indicar o tempo máximo de espera e os locais em que haveria elementos que esperam esse tempo (2.5).

Formalização

Cenário 2.1 & 2.2 - determinar encaminhamento para um grupo dada a sua dimensão

- Input:
  - dataset com informação sobre os nós (N) e arestas (E)
- Output:
  - Tamanho do grupo (G)
  - número máximo de passageiros na viagem (C)
  - número de pessoas que não conseguiram transporte (N = G - C)
  - respetivo percurso (P)
- Variáveis de decisão:
  - G : Tamanho do grupo
- Funções-objetivo:
  - maximizar C (número máximo de passageiros na viagem)
- Restrições e Domínios de valores:
  - $1 \leq G$, $1 \leq C$, para todo o $G, C \in \mathbb{N}$
  - $0 \leq N < G$ para todo o $N \in \mathbb{N}$
- Objetivos:
  - encontrar um encaminhamento P tal que C = G

Algoritmos relevantes

Cenário 2.1 & 2.2 - determinar encaminhamento para um grupo dada a sua dimensão

- O algoritmo usado foi uma variação do algoritmo de Edmonds-Karp usado no cenário 2.3, com uma limitação ao fluxo máximo.
- Isto permite-nos escolher o melhor caminho para um grupo de tamanho específico, por oposição ao melhor caminho para o maior grupo possível.

```cpp
void EdmondsKarpGraph::edmondsKarpFixedFlow(int group_size) {
    int flow;
    do {
        unvisitAll();
        flow = bfs();
        maxFlow += flow;
        if(maxFlow >= group_size){
            break;
        }
    } while (flow != 0);
}
```

```cpp
int EdmondsKarpGraph::bfs() {
    unvisitAll();

    // queue of unvisited nodes
    queue<int> q;
    q.push(this->s); visit(s);

    vector<Edge*> aux( n: t+1);

    // while there are still unvisited nodes
    while(!q.empty()){
        int currentNode = q.front(); q.pop();

        // if current node is target node
        if(currentNode == t) break;

        // for each adjacent edge
        for(Edge &edge : nodes[currentNode].adj){
            int cap = edge.residualCap();

            // if there is still capacity left and next node wasn't visited
            if(cap > 0 && !isVisited(edge.to)){
                aux[edge.to] = &edge;
                q.push(edge.to); // add to queue
                visit(edge.to); // mark as visited
            }
        }
    }
```

Análise de complexidade

Cenário 2.1 & 2.2 - determinar encaminhamento para um grupo dada a sua dimensão

Complexidade temporal:

- O algoritmo utilizado apresenta complexidade temporal O(V * E^2), que se deve ao uso do Método de Edmonds-Karp.

Complexidade espacial:

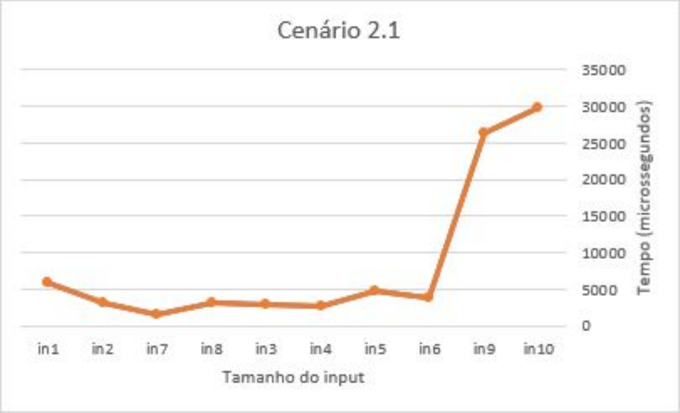
- A complexidade espacial associada é S(V,E) = O(V + E), presente no uso de vetores e de uma queue auxiliares e inerentes ao uso do algoritmo.

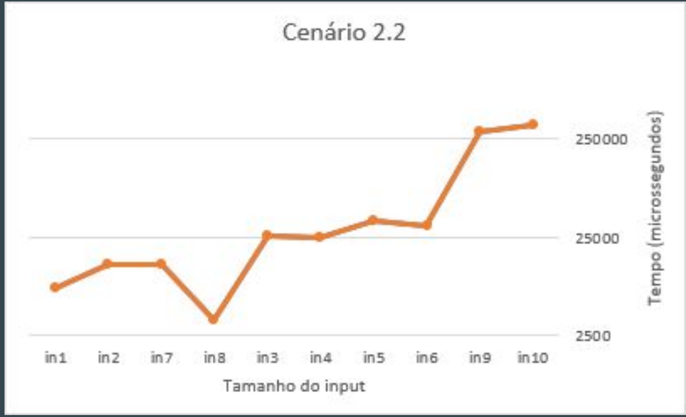
Nota: V - número de nós (locais)

E - número de ramos (veículos)

Resultados da Avaliação Empírica

Cenário 2.1 & 2.2 - determinar encaminhamento para um grupo dada a sua dimensão





Formalização

Cenário 2.3 - dimensão máxima de um grupo e um encaminhamento

- Input:
  - dataset com informação sobre os nós (N) e arestas (E)
- Output:
  - Tamanho do grupo (G)
  - número máximo de passageiros na viagem (C)
  - número de pessoas que não conseguiram transporte ($N = G - C$)
  - respetivo percurso (P)
- Variáveis de decisão:
  - G : Tamanho do grupo
- Funções-objetivo:
  - maximizar C (número máximo de passageiros na viagem)
- Restrições e Domínios de valores:
  - $1 \leq G, 1 \leq C$, para todo o $G, C \in \mathbb{N}$
  - $0 \leq N < G$ para todo o $N \in \mathbb{N}$
- Objetivos:
  - encontrar um encaminhamento P tal que $C = G$

Algoritmos relevantes

Cenário 2.3 - dimensão máxima de um grupo e um encaminhamento

- O algoritmo usado para a resolução deste problema foi o algoritmo de Edmonds-Karp que, na sua essência, utilizando uma pesquisa em largura, encontra os caminhos mais curtos num grafo e calcula o fluxo máximo desse caminho.
- Após calcular o fluxo máximo de cada um dos caminhos existentes, o fluxo máximo do grafo inteiro é calculado, obtendo-se assim o número máximo de pessoas que poderiam ser transportadas na rede.

```cpp
void EdmondsKarpGraph::edmondsKarp() {
    int flow;
    do {
        unvisitAll();
        flow = bfs();
        maxFlow += flow;
    } while (flow != 0);
}
```

```cpp
int EdmondsKarpGraph::bfs() {
    unvisitAll();

    // queue of unvisited nodes
    queue<int> q;
    q.push(this->s); visit(s);

    vector<Edge*> aux( n: t+1);

    // while there are still unvisited nodes
    while(!q.empty()){
        int currentNode = q.front(); q.pop();

        // if current node is target node
        if(currentNode == t) break;

        // for each adjacent edge
        for(Edge &edge : nodes[currentNode].adj){
            int cap = edge.residualCap();

            // if there is still capacity left and next node wasn't visited
            if(cap > 0 && !isVisited(edge.to)){
                aux[edge.to] = &edge;
                q.push(edge.to); // add to queue
                visit(edge.to); // mark as visited
            }
        }
    }
```

Análise de complexidade

Cenário 2.3 - dimensão máxima de um grupo e um encaminhamento

Complexidade temporal:

- O algoritmo utilizado apresenta complexidade temporal O(V * E^2), que se deve ao uso do Método de Edmonds-Karp.

Complexidade espacial:

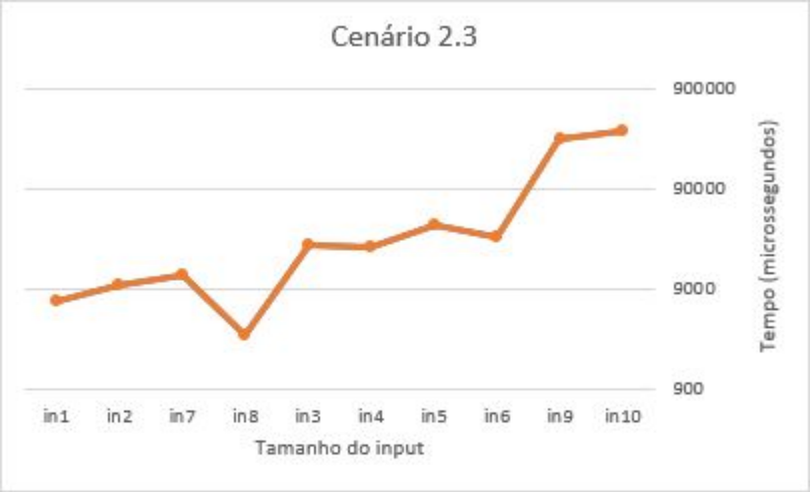
- A complexidade espacial associada é S(V,E) = O(V + E), presente no uso de vetores e de uma queue auxiliares e inerentes ao uso do algoritmo.

Nota: V - número de nós (locais)

E - número de ramos (veículos)

Resultados da Avaliação Empírica

Cenário 2.3 - dimensão máxima de um grupo e um encaminhamento



Formalização

Cenário 2.4 - determinar quando é que o grupo se reuniria novamente

- Input:
    - dataset com informação sobre os nós (N) e arestas (E)
- Output:
    - duração mínima de espera para que o grupo se junte novamente
- Variáveis de decisão:
    - tempo que um grupo leva a reunir-se de novamente (durMin)
- Funções-objetivo:
    - maximizar (N, i=1) (durMin)
- Restrições e Domínios de valores:
    - $0 \leq durMin$
    - $2 \leq N$
    - $1 \leq E$
- Objetivos:
    - determinar quanto tempo depois do início é que o grupo se reuniria novamente

Algoritmos relevantes

Cenário 2.4 - determinar quando é que o grupo se reuniria novamente

- O objetivo do subproblema é determinar quando é que um grupo se voltaria a reunir.
- Para tal, usamos o algoritmo Critical Path, calculando o caminho crítico e o Early Start para cada node
- A ideia para determinar o tempo que um grupo levaria a voltar a estar junto passa por iterar sobre os nodes do grafo, calculando o Early Start para cada um dos seus adjacentes e reduzindo o grau destes nodes adjacentes para disponibilizá-los para serem iterados quando o seu grau for igual a zero.
- De referir que, à medida que iteramos sobre nodes, devemos guardar (numa variável, no nosso caso durMin) o valor correspondente ao seu Early Start caso este seja superior ao anterior valor guardado.
- No final, o valor do retorno deverá ser o do durMin, a resposta ao problema em questão.

```
while(!q.empty()){
    int v = q.front(); q.pop();
    if(durMin < ES[v]){
        durMin = ES[v];
        vf = v;
    }
    for(Edge edge : nodes[v].adj){
        //if(edge.flow == 0) continue;
        int w = edge.to;
        if(ES[w] < ES[v] + edge.dur){
            ES[w] = ES[v] + edge.dur;
            Prec[w] = v;
        }
        GrauE[w]--;
        if(GrauE[w] == 0){
            q.push(w);
        }
    }
}
```

Análise de complexidade

Cenário 2.4 - determinar quando é que o grupo se reuniria novamente

Complexidade temporal:

- O algoritmo utilizado apresenta complexidade temporal O(V * E).

Complexidade espacial:

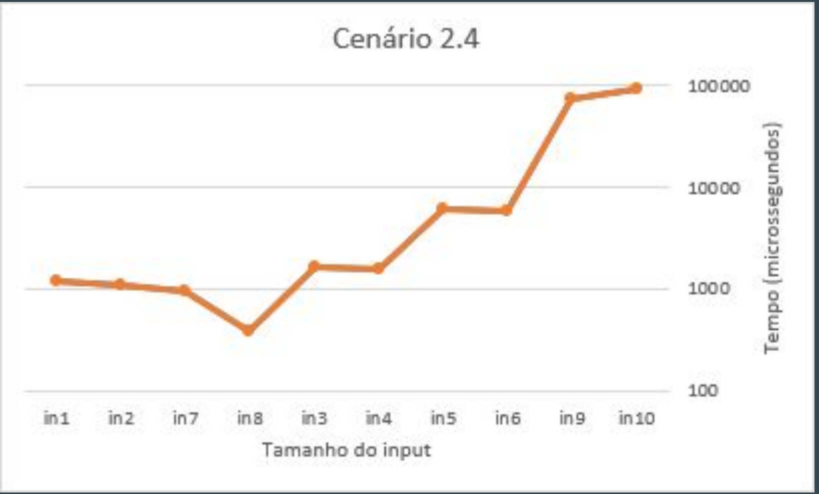
- A complexidade espacial associada é S(V,E) = O(V + E), presente no uso de vetores e de uma stack auxiliares e inerentes ao uso do algoritmo.

Nota: V - número de nós (locais)

E - número de ramos (veículos)

Resultados da Avaliação Empírica

Cenário 2.4 - determinar quando é que o grupo se reuniria novamente



Formalização

Cenário 2.5 - indicar o tempo máximo de espera e os locais em que haveria elementos que esperam esse tempo

- Input:
  - dataset com informação sobre os nós (N) e arestas (E)
- Output:
  - Tempo máximo de espera
  - Locais onde há tempo máximo de espera
- Variáveis de decisão:
  - tempo de espera (T) - Folga Total
- Funções-objetivo:
  - maximizar (N, i=1) (LF - ES - DUR)
  - ou seja, maximizar T
- Restrições e Domínios de valores:
  - $2 \leq N$
  - $1 \leq E$
  - $0 \leq DUR$
- Objetivos:
  - determinar o tempo máximo de espera
  - determinar os locais onde se espera esse tempo

(LF) - Latest Finish
(ES) - Earliest Start
(DUR) - duração mínima de espera

Algoritmos relevantes

Cenário 2.5 - indicar o tempo máximo de espera e os locais em que haveria elementos que esperam esse tempo

- O subproblema em questão tem por objetivo determinar qual será o tempo máximo de espera de um ou mais nodes do grafo bem como em quais nodes isto ocorre.

- Para tal, recorremos ao uso do algoritmo Critical Path, desta vez calculando o Latest Finish, mas também usando o do Early Start do 2.4. Usamos o Early Start para conseguirmos utilizar o valor deste para cada node bem como o valor da DUR (definição no slide 26).

- Para calcularmos o Latest Finish de um nó, é necessário criar o grafo transposto do originalmente usado em 2.4, pelo que temos de o criar.

- Após a sua criação, iteramos sobre os nodes e os seus respetivos adjacentes, reduzindo os seus graus (caso o grau seja zero, adicionamos à pilha para ser iterado) e, caso se verifique, alterando o respetivo valor das suas Latest Finish.

- No final, analisamos todos os nodes com vista a determinar qual o(s) local/locais onde o tempo de espera é maior (o tempo de espera máximo é calculado com base na *Folga Total**)

*Folga Total = LatestFinish - EarlyStart - Duration

Algoritmos relevantes

Cenário 2.5 - indicar o tempo máximo de espera e os locais em que haveria elementos que esperam esse tempo

```cpp
void EdmondsKarpGraph::latestFinish() {
    criticalPath();

    stack<int> stack;
    vector<Node> nodes2( n: t + 1);

    for (int i = 1; i <= t; i++) {
        LF[i] = durMin;
        GrauS[i] = 0;
```

```cpp
    while (stack.size() != 0) {
        int v = stack.top();
        stack.pop();

        for (auto &w: nodes2[v].adj) {
            if (LF[w.to] > (LF[v] - w.dur)) {
                LF[w.to] = LF[v] - w.dur;
            }

            GrauS[w.to]--;

            if (GrauS[w.to] == 0) stack.push(w.to);
        }
    }
```

```cpp
//biggest FT = LF - ES - duracao
for (int i = 1; i <= t; i++) {
    if (nodes2[i].adj.size() != 0) {
        for (auto e: nodes2[i].adj) {
            if (e.dur == 0) continue;

            int temp_FT = LF[i] - ES[e.to] - e.dur;

            if (temp_FT > FT) {
                FT = temp_FT;
                FT_nodes.clear();
                FT_nodes.push_back(i);
            } else if (temp_FT == FT && !inVector( FT_nodes: FT_nodes, i)) {
                FT_nodes.push_back(i);
            }
        }
    }
}

cout << "Tempo Maximo de Espera: " << FT << endl;
cout << "Locais onde ocorre esta espera: ";

for (auto e: FT_nodes) cout << e << " ";

cout << endl;
```

Análise de complexidade

Cenário 2.5 - indicar o tempo máximo de espera e os locais em que haveria elementos que esperam esse tempo

Complexidade temporal:

- O algoritmo utilizado apresenta complexidade temporal O(V * E).

Complexidade espacial:

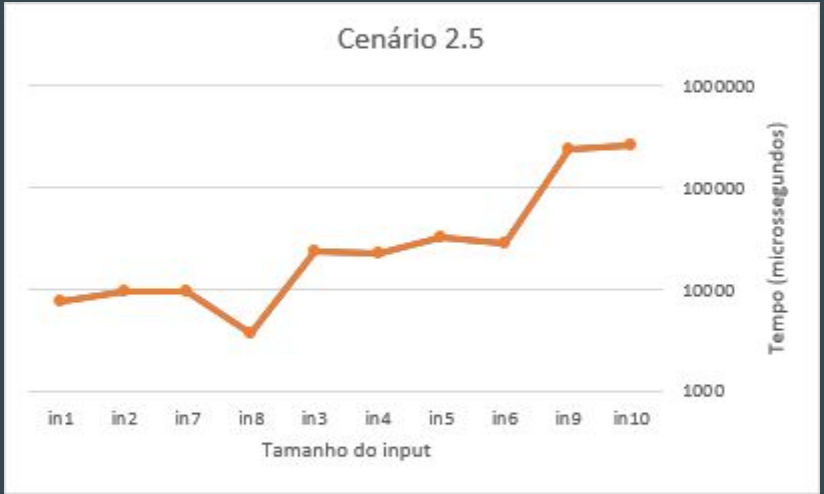
- A complexidade espacial associada é S(V,E) = O(V), presente no uso de vetores e de uma stack auxiliares e inerentes ao uso do algoritmo.

Nota: V - número de nós (locais)

E - número de ramos (veículos)

Resultados da Avaliação Empírica

Cenário 2.5 - indicar o tempo máximo de espera e os locais em que haveria elementos que esperam esse tempo



Dificuldades/Esforço

Dificuldades:

- As principais dificuldades deste trabalho foram a conceção dos algoritmos para a realização dos cenários.
- Correção de bugs.

Divisão do esforço:

- Gustavo Costa (up202004187) - 33.(3)%
- José Araújo - (up202007921) - 33.(3)%
- Ricardo Cavalheiro (up202005103) - 33.(3)%