trabalho DA:
Gestão de distribuição de clientes em uma agência de viagens

T06_G62

Diogo Babo up202004950
João Oliveira up202004407
João Pinheiro up202008133

O problema proposto:

“Com a saturação dos serviços de entregas, os acionistas decidiram diversificar o ramo de negócio e apostar também na promoção de viagens turísticas. A empresa tem a sua própria frota de veículos. Cada um fará um único trajeto, com uma certa capacidade, custo de bilhete, origem e destino.

Pretende-se um sistema capaz de apoiar a gestão de pedidos para transporte de grupos de pessoas."

Cenário 1 - “Grupos que não se separam”

- Maximizar a dimensão do grupo e indicar um qualquer encaminhamento.
- Maximizar a dimensão do grupo e minimizar o número de transbordos, apresentar as soluções pareto-ótimas.

Cenário 1.1 - Formalização

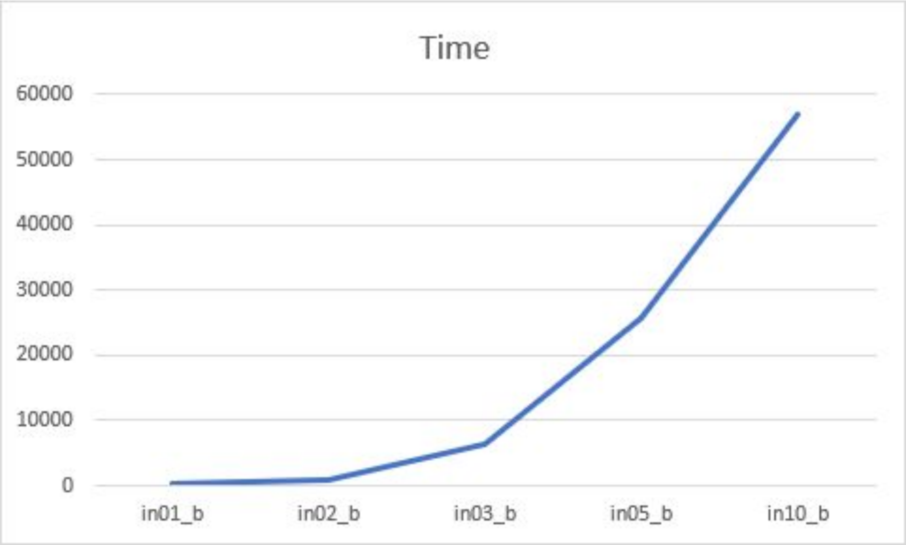
- **Dados:** dataset com nós e respectivas arestas
- **Output:** dimensão máxima do grupo, percurso do mesmo e número de transbordos
- **Variáveis de decisão:** dimensão do grupo e número de transbordos
- **Função-objetivo:** maximizar a dimensão do grupo
- **Restrições e Domínios de valores:**
- N >= 1 & E >= 0;
- T & D >= 0.
- **Objetivo:** maximizar a dimensão do grupo e indicar um encaminhamento

Algoritmo 1.1:
(Capacidade Máxima)

- O algoritmo ;
- Time complexity: O((V + E) * log2 (V));
- Space complexity: O(V + E) .

```
void Graph::MaxCapWays(int s) {
    for (int v=1; v<=n; v++) { nodes[v].parent = -1; nodes[v].distance = 0; }
    nodes[s].distance = INT_MAX / 2;

    MaxHeap<int, int> heap(n, notFound: -1);
    heap.insert( key: s, value: nodes[s].distance);
    for(int v=1;v<=n; v++) heap.insert( key: v, value: nodes[v].distance);

    while(heap.getSize()>0) {
        int v = heap.removeMax();

        Node* u = &nodes[v];
        for(const Edge &e: u->adj){
            if(!heap.hasKey( key: e.dest)) continue;
            if(nodes[e.dest].distance < min(u->distance, e.capacity)){
                nodes[e.dest].parent=v;
                nodes[e.dest].distance=e.capacity;
                heap.increaseKey( key: e.dest, value: e.capacity);
            }
        }
    }
}
```

Algoritmo 1.1:
(Avaliação Empírica)



Cenário 1.2 - Formalização

- **Dados:** dataset com nós e respectivas arestas
- **Output:** dimensão máxima do grupo, e apresentar soluções pareto-ótimas
- **Variáveis de decisão:** dimensão do grupo
- **Função-objetivo:** maximizar a dimensão do grupo
- **Restrições e Domínios de valores:**
- N >= 1 & E >= 0;
- T & D >= 0.
- **Objetivo:** maximizar a dimensão do grupo, minimizar transbordos e indicar soluções pareto-ótimas

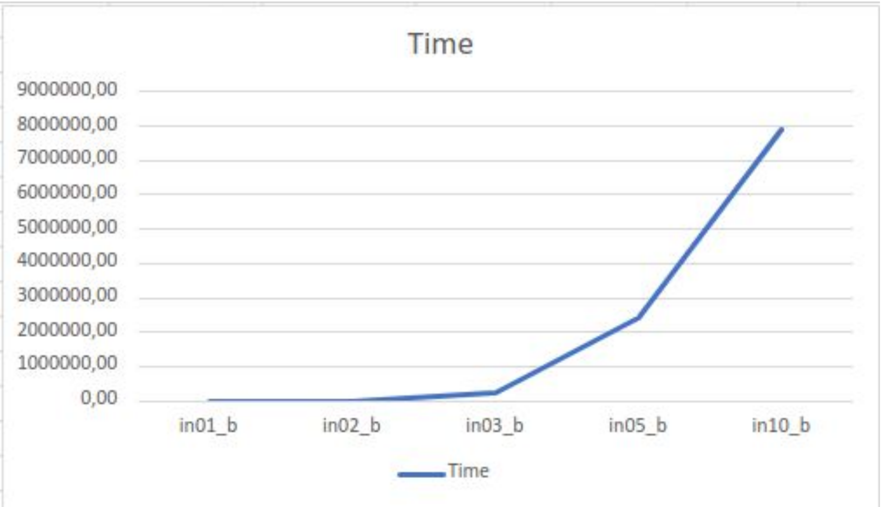
Algoritmo 1.2: (Pareto-Ótimas)

- O algoritmo ;
- Time complexity: O(V+ E)* log2 V^2));
- Space complexity: O((V-1)*V^2).

```cpp
void Empresa::one2(int s, int t) {
    Graph temp = rede;
    rede.distance( a: s,  b: t);

    vector<int> path;
    vector<int> d = rede.getDistances();

    vector<pair<vector<int>,int>> solutions;
    vector<int> pathAux;
    vector<int> capsAux;

    int transbordos = d[t];

    rede.getPath( pVector: &path, t);
    int pathCap = rede.pathCapacity( vector1: path);

    rede.setPath( vector1: path);
    solutions.push_back(pair<vector<int>,int>( x: path,  y: pathCap));
    while(rede.unusedNodes()>1){
        rede.MaxCapWaysWithUse(s);
        pathAux.clear();
        rede.getPath( pVector: &pathAux, t);
        if(solutions.back().first != pathAux){
            solutions.push_back(pair<vector<int>,int>( x: pathAux,  y: rede.pathCapacity( vector1: pathAux)));
        }
        rede.setUseNode();
    }

    sort( first: solutions.begin(),  last: solutions.end(),  comp: sortPair);

    reverse( first: solutions.begin(),  last: solutions.end());
    pair<vector<int>,int> last;
```

```cpp
if(solutions.size()>0){
    last = solutions[0];
}
for(int x = 1; x<solutions.size();x++){
    pair<vector<int>,int> actual = solutions[x];
    if(last.first.size()<actual.first.size() && last.second<actual.second){
        last = actual;
        continue;
    }
    if(last.first.size()==actual.first.size() && last.second<=actual.second){
        last = actual;
        continue;
    }
    solutions.erase( position: solutions.begin()+x);
    x--;
}

cout << "1.2) Mínimo de transbordos entre " << s << " para " << t << ": " << transbordos << endl;
cout << "Dimensão máxima do grupo (sem se separar): " << pathCap << endl;
cout << path[0];
for (int i = 1; i < path.size(); i++) {
    cout << " -> "<< path[i];
}
cout << endl;
for(int z = 1; z <solutions.size();z++){
    cout << endl << "Outra parento-solução: "<< endl;
    cout << "Numero de trasnbordos : "<<solutions[z].first.size() - 1 << endl;
    cout << "Capacidade do grupo :"<< solutions[z].second << endl;
    cout << solutions[z].first[0];
    for (int i = 1; i < solutions[z].first.size(); i++) {
        cout << " -> "<< solutions[z].first[i];
    }
    cout << endl;
}
```

Algoritmo 1.2: (Avaliação Empírica)



Cenário 2 - “Grupos que podem separar-se”

- Determinar um encaminhamento para um grupo, dada a sua dimensão;
- Corrigir um encaminhamento, se necessário, para um determinado aumento da dimensão do grupo;
- Determinar a dimensão máxima do grupo e um encaminhamento;
- Partindo de um encaminhamento, determinar quando é que o grupo se volta a reunir, no mínimo;
- Nas mesmas condições, indicar o tempo máximo de espera e os locais em que houve espera.

Algoritmo Edmonds-Karp:

- Implementação do Ford-Fulkerson utilizando uma BFS, esta que escolhe um encaminhamento tendo em conta o menor número de arestas.
- Time complexity: $O(E*V^3)$;
- Space complexity: $O(2(V+E))$.

```cpp
int Graph::fordFulkerson(Graph& residual, int s, int t, vector<vector<int>> *paths, int dimension) {
    int max_flow = 0;

    while (true) {
        residual.bfshelper( s);
        if(residual.nodes[t].parent == -1) break;

        int path_flow = INT_MAX/2;
        for (int v = t; v != s; v = residual.nodes[v].parent) {
            int u = residual.nodes[v].parent;
            for (auto e : residual.nodes[u].adj) {
                if (e.dest == v) path_flow = min(path_flow, e.capacity);
            }
        }

        for (int v = t; v != s; v = residual.nodes[v].parent) {
            int u = residual.nodes[v].parent;
            for (auto &e : residual.nodes[u].adj) {
                if (e.dest == v) e.capacity -= path_flow;
            }
            for (auto &e : residual.nodes[v].adj) {
                if(e.dest == u) e.capacity += path_flow;
            }
        }

        vector<int> path;
        residual.getPath( &path, t);
        paths->push_back(path);
        max_flow += path_flow;
        if (dimension != -1 && max_flow >= dimension) return max_flow;
    }

    return max_flow;
}
```

Cenário 2.1 & 2.2 - Formalização

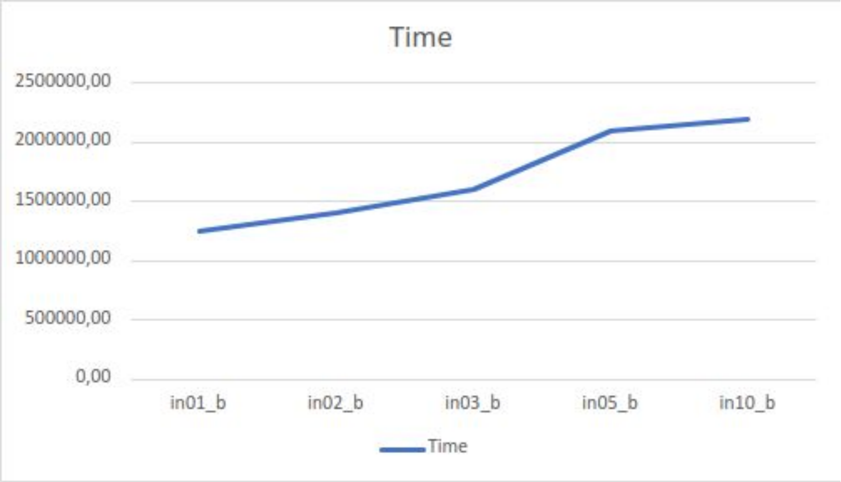
- **Dados:** dataset com nós e respectivas arestas, capacidade do grupo
- **Output:** todos os percursos possíveis para o respectiva dimensão do grupo
- **Variáveis de decisão:** dimensão do grupo
- **Função-objetivo:** maximizar a dimensão do grupo
- **Restrições e Domínios de valores:**
- $N >= 1$ & $E >= 0$;
- $DG > 0$;
- T & D >= 0.
- **Objetivo:** indicar encaminhamentos para a dimensão do grupo

Algoritmo 2.1:
(Encaminhamento Grupo)

- O algoritmo ;
- Time complexity: O($E*V^3$);
- Space complexity: O(2(V+E)) .

```cpp
void Empresa::two1(int s, int t) {
    Graph temp = rede;
    int dimension, plus;
    cout << "2.1) Size of the group : \n";
    cin >> dimension;

    vector<vector<int>> paths;
    Graph residual = rede.createResidual();
    int maxFlow = Graph::fordFulkerson( residual, s, t, &paths, dimension);

    if (maxFlow < dimension) cout << "Can only transport " << maxFlow << " persons;" << endl;

    for (auto path : paths) {
        cout << path[0];
        for (int i = 1; i < path.size(); i++) {
            cout << " -> "<< path[i];
        }
        cout << endl;
    }
```

Algoritmo 2.1 & 2.2: (Avaliação Empírica)



Algoritmo 2.2:
(Corrigir Encaminhamento)

- O algoritmo ;
- Permitir ao utilizador incrementar o tamanho do grupo, e caso seja necessário corrigir os encaminhamentos.
- Time complexity: $O(EV^3)$;
- Space complexity: $O(2(V+E))$.

```
cout << "2.2) Add to the group x unities : x -> ";
cin >> plus;

int maxCap = rede.checkMaxCap(paths);
paths.clear();
if (maxCap < dimension + plus) {
    residual = rede.createResidual();
    Graph::fordFulkerson( residual, s, t, &paths, dimension + plus);
}

if(!paths.empty()) cout << "Correção de encaminhamento para maior dimensão do grupo: " << endl;
else cout << "Não foi necessária correção de encaminhamento" << endl;
for (auto path : paths) {
    cout << path[0];
    for (int i = 1; i < path.size(); i++) {
        cout << " -> "<< path[i];
    }
    cout << endl;
}
```

Cenário 2.3 - Formalização

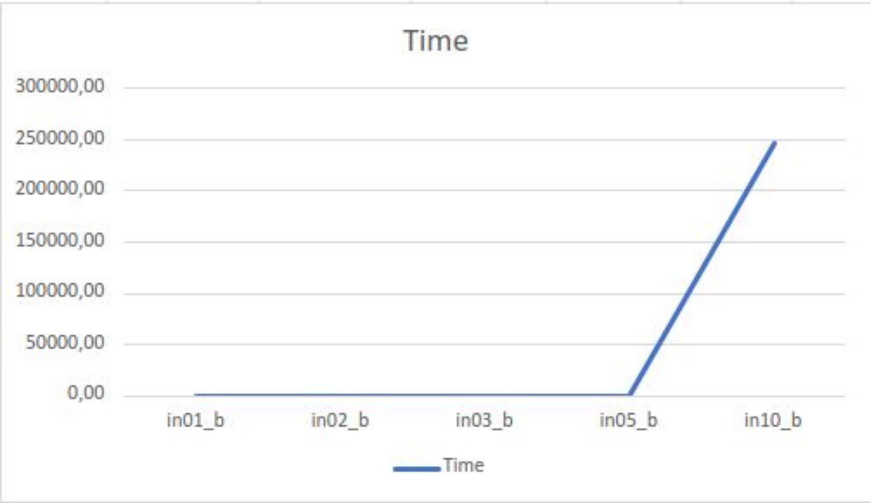
- **Dados:** dataset com nós e respectivas arestas
- **Output:** fluxo máximo de um grupo, encaminhamentos
- **Variáveis de decisão:** dimensão do grupo
- **Função-objetivo:** maximizar a dimensão do grupo
- **Restrições e Domínios de valores:**
- N >= 1 & E >= 0;
- T & D >= 0.
- **Objetivo:** indicar encaminhamentos tal que a dimensão do grupo seja máxima

Algoritmo 2.3:
(Dimensão Máxima Grupo)

- O algoritmo ;
- Time complexity: $O(EV^3)$;
- Space complexity: $O(2(V+E))$.

```cpp
void Empresa::two3(int s, int t) {
    Graph temp = rede;
    Graph residual = this->rede.createResidual();
    vector<vector<int>> paths;

    int max_flow = Graph::fordFulkerson( residual, s, t, &paths);
    cout << "2.3) Ford-Fulkerson max_flow : " << max_flow << endl;

    int max_time = 0;
    for (auto path : paths) {
        int time = this->rede.getTime( path);
        max_time = max(max_time, time);

        cout << path[0];
        for (int i = 1; i < path.size(); i++) {
            cout << " -> "<< path[i];

        }
        cout << endl;
    }
    rede = temp;
}
```

Algoritmo 2.3: (Avaliação Empírica)



Cenário 2.4 - Formalização

- **Dados:** dataset com nós e respectivas arestas
- **Output:** duração mínima para o grupo se reunir no destino
- **Variáveis de decisão:** duração mínima
- **Função-objetivo:** minimizar o tempo
- **Restrições e Domínios de valores:**
- N >= 1 & E >= 0;
- DuraçãoMin >= 0;
- T & D >= 0.
- **Objetivo:** obter o tempo mínimo para que todo o grupo se reúna no destino

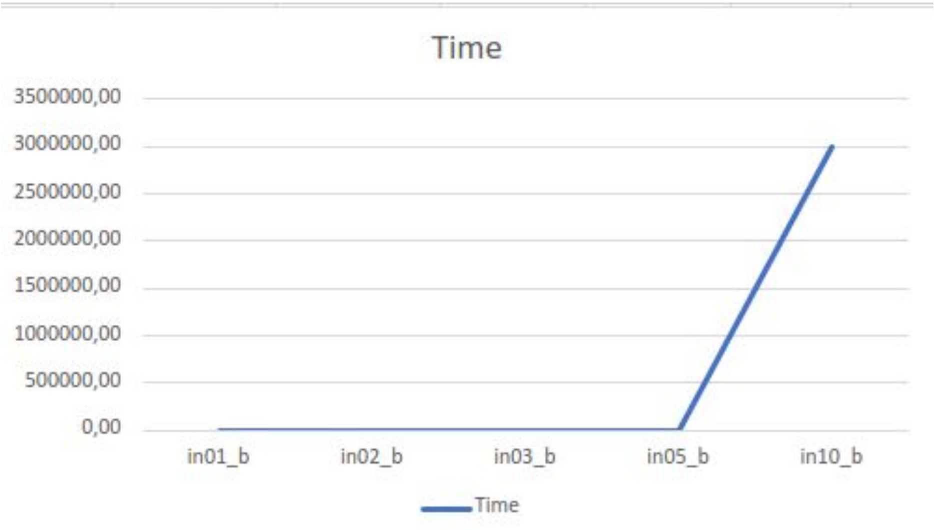
Algoritmo 2.4:
(Tempo min. reunir destino)

- Funciona apenas para os encaminhamentos anteriores como para o grafo inteiro. (ambas opções dadas ao utilizador).
- Calcular o Earliest Start para cada nó e a duração minima para o grupo chegar ao destino.
- Time complexity: O( );
- Space complexity: O( ) .

```cpp
int Graph::minDuration(int s,int t) {
    Graph residual = this->createResidual();

    vector<vector<int>> paths;
    vector<pair<int,int>> waitTime;
    fordFulkerson( residual, s, t, &paths);

    for(auto &x :residual.nodes) {
        for(auto &y : x.adj) {
            y.flow = 0;
        }
    }
    for(auto x : paths) {
        for(int i = 0; i < x.size() - 1; i++) {
            int from = x[i];
            int to = x[i+1];
            for(int j = 0; j < residual.nodes.size(); j++) {
                for(auto &w : residual.nodes[j].adj) {
                    if(j == from && to == w.dest) {
                        w.flow = 1;
                    }
                }
            }
        }
    }
```

```cpp
while(!q.empty()) {
    int v = q.front();
    q.pop();
    if(durMin < ES[v]) {
        durMin = ES[v];
        vf = v;
    }
    for(auto e : residual.nodes[v].adj) {
        int w = e.dest;
        if(e.flow == 0) {
            continue;
        }
        if(ES[w] < ES[v] + e.weight) {
            ES[w] = ES[v] + e.weight;
            Pred[w] = v;
        }
        Grau[w]--;
        if(Grau[w] == 0) {
            q.push( w);
        }
    }
}
return durMin;
```

Algoritmo 2.4: (Avaliação Empírica)



Cenário 2.5 - Formalização

- **Dados:** dataset com nós e respectivas arestas
- **Output:** tempo máximo de espera e os nós em que este ocorreu
- **Variáveis de decisão:** Folga Total, Duração Minima
- **Função-objetivo:** maximizar a Folga Total
- **Restrições e Domínios de valores:**
- N >= 1 & E >= 0;
- DuraçãoMin >= 0;
- T & D >= 0.
- **Objetivo:** obter o tempo máximo de espera, e os locais(nós) em que este aconteceu

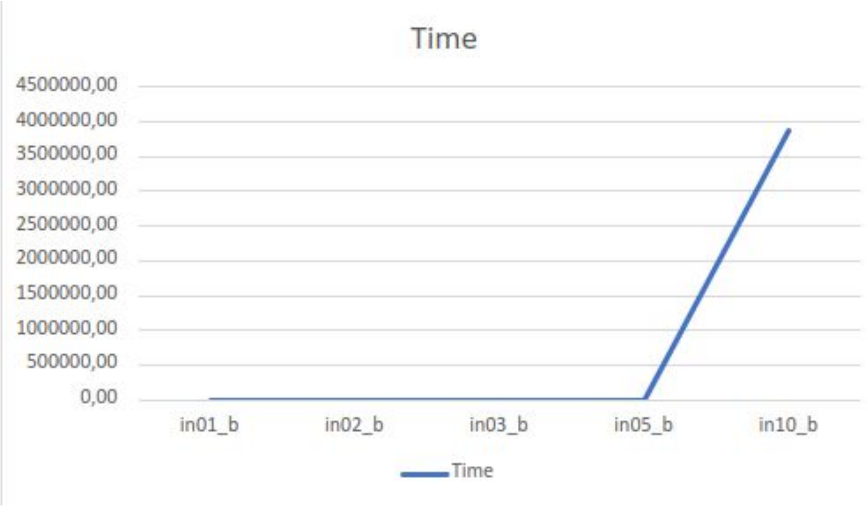
Algoritmo 2.5:
(Tempo máximo espera)

- Funciona apenas para os encaminhamentos anteriores como para o grafo inteiro. (ambas opções dadas ao utilizador).
- Utilizar o algoritmo 2.4 para obter ES e duração Minima.
- Calcular a Folga Total para cada nó.
- Time complexity: O( );
- Space complexity: O( ) .

```cpp
for(auto &x :this->nodes) {
    for(auto &y : x.adj) {
        y.flow = 0;
    }
}
for(auto x : paths) {
    for(int i = 0; i < x.size() - 1; i++) {
        int from = x[i];
        int to = x[i+1];
        for(int j = 0; j < this->nodes.size(); j++) {
            for(auto &w : this->nodes[j].adj) {
                if(j == from && to == w.dest) {
                    w.flow = 1;
                }
            }
        }
    }
}

Graph graphT = createTranspose();
```

```cpp
for(auto &x : LF) {
    x = durMin;
}

while(!q.empty()) {
    int v = q.front();
    q.pop();
    for(auto &e : graphT.nodes[v].adj) {
        int w = e.dest;
        if(e.flow == 0) {
            continue;
        }
        if(LF[w] > LF[v] - e.weight) {
            LF[w] = LF[v] - e.weight;
        }
        Grau[w]--;
        if(Grau[w] == 0) {
            q.push(w);
        }
    }
}

vector<int> temp;
for(int i = 1; i < nodes.size(); i++) {
    for(auto x : nodes[i].adj) {
        if(x.flow == 0) {
            continue;
        }
        if(LF[x.dest] - x.weight - ES[i] > FT[x.dest]) {
            FT[x.dest] = LF[x.dest] - x.weight - ES[i];
        }
    }
}
```

Algoritmo 2.5:
(Avaliação Empírica)



Limitações e possíveis melhorias:

1. Cenário 1.2;
2. Cenário 2.5;

Obrigado pela atenção

Diogo Babo 33.3%
João Oliveira 33.3%
João Pinheiro 33.3%