U.PORTO
FEUP FACULDADE DE ENGENHARIA
UNIVERSIDADE DO PORTO

Agência de Viagens

Um grafo de ligações via transportes públicos entre países

Grupo H – Turma 3 – MIEIC 2º ano
Professor Rosaldo Rossetti

Autoria:
Gil D. M. Teixeira – 201505735 – up201505735@fe.up.pt
Ricardo J. S. Pereira – 201503716 – up201503716@fe.up.pt
Rúben J. S. Torres – 201405612 – up201405612@fe.up.pt

Introdução

Este trabalho tem como objetivo a criação de um sistema que permita encontrar um caminho ótimo entre duas localizações (no mesmo país ou entre dois países), utilizando para isso um grafo que represente estas ligações, sobre o qual será executado um algoritmo – ou vários – com intuito de resolver este problema. Posteriormente, pretendemos estender a funcionalidade do programa para encontrar um caminho ótimo dadas várias localizações que o utilizador pretenda visitar.

O grafo é constituído por nós e arestas; estes, por sua vez, representam as localizações e as ligações entre estas, respetivamente. Os nós contêm informação tal como o nome da localização, um vetor de arestas para as ligações que partem desta e ainda uma classe com informação relativa ao alojamento nessa localização. As arestas por sua vez contêm informação relativa ao custo da viagem com diferentes meios de transporte, a distância entre os nós que ligam e o tempo médio da viagem para cada meio de transporte.

Descrição do problema

Por caminho ótimo, referimo-nos ao caminho de menor custo para o utilizador do sistema, que por norma é também o caminho mais curto (e de menor duração). Uma vez, no entanto, que a ligação entre os sítios pode não ser direta (requer um ou mais pontos intermédios), há que ter em conta múltiplas formas de chegar ao destino. Adicionalmente, temos ainda que considerar que as viagens entre estes pontos intermédios podem não ser imediatas, obrigando o utilizador a esperar nestes pontos, e em certos casos a adquirir alojamento enquanto espera pela viagem. Portanto devemos também ter em conta o custo deste, que pode variar consoante a época e os diferentes dias da semana. No caso de o utilizador pretender visitar mais que uma localização, todas estas considerações relativas aos pontos intermédios aplicam-se automaticamente, requerendo no entanto uma especialização do algoritmo para garantir que os nós correspondentes às localizações selecionadas são percorridos pelo menos uma vez (e idealmente, na maior parte dos casos, no máximo uma vez). Para todos estes casos, utilizaremos uma versão modificada do algoritmo de Dijkstra, juntamente com um algoritmo simples para calcular a menor distância entre dois nós (sem considerar pesos) de modo a tornar o cálculo mais rápido.

Casos de utilização:

- Criação de uma Agência de Viagens.
- Leitura de ficheiros de dados.
- Calcular a viagem mais económica para um destino.
- Calcular a viagem mais económica para vários destinos.
- Visualizar grafo.

Dados de entrada:

- Cidade em que se pretende iniciar a viagem;
- Cidade(s) que pretende(m) visitar;
- Data em que se pretende iniciar a viagem;

- Número de dias de alojamento(opcional);
- Transportes que não se pretende utilizar.

Restrições:

- Destino e origem têm de ser diferentes;
- Data tem de ser válida;

Resultados esperados:

Transportes a usar com a indicação da origem, destino, tipo de transporte, preço e duração de todas as viagens, e custo do alojamento.

Solução implementada

Classes usadas:

- A agência de viagens foi implementada usando a Classe **TravelAgency** que contém o grafo *travelAgencyGraph* e vetor de strings *transportTypes*.
- O grafo é composto por vértices e arestas a unir os mesmos.
- A informação dos vértices é representada pela classe **Node**, e a das arestas pela classe **Weight**.
- A classe **Node** é composta pelo *nodeID* (identificador do nó), *cityName* (nome da cidade que representa), *accomodation*(alojamento) e *latitude* e *longitude* da cidade.
- A classe **Accommodation** é composta pelo *price* (preço base), vetor *datesPriceChange* (vetor de pares de datas em que o preço do alojamento é alterado), *percentageToChange*(vetor que representa a percentagem em que preço é alterado relativamente à da data) e *percentageDay* (*map* que contém as constantes que são multiplicadas pelo preço base de acordo com o dia da semana).
- A classe **Weight** é composta por um vetor *allTrips*, composto por elementos da classe **TripInfo**.
- A classe **TripInfo** é composta por *type* (o tipo de transporte usado, por exemplo, avião), *price_in_euros* (preço da viagem) e *time_in_minutes* (tempo da viagem).
- Além das classes acima descritas, é usada, como auxiliadora, a classe *Data*, que tem como membros: *ano*, *mês* e *dia*.

Funcionamento do programa:

O programa começa por ler 2 ficheiros “Nodes.txt” e “Edges.txt”. Se for encontrada alguma irregularidade, é lançada uma exceção; se não, a informação é armazenada no grafo *travelAgencyGraph*.

Depois, é chamada a função *mainMenu*, que mostra ao utilizador, na forma de um menu, as funcionalidades do programa.

Ficheiros de texto:

Os 2 ficheiros de texto começam com um cabeçalho a indicar o tipo de ficheiro (exemplo: “#Edges_start”, “#Vertex_start”) e terminam com o rodapé “#End”.

O ficheiro de **Arestas** guarda a informação na forma “ID da aresta ; ID do nó Origem ; ID do nó Destino ; tipo de transporte ; preço da viagem ; duração da viagem em minutos”, sendo que os últimos 3 elementos podem ser repetidos. Exemplo: “7;4;1;Ship;5;180;Airplane;50;100”.

O ficheiro de **Nós** guarda a informação na forma “ID do Nó ; nome da cidade; preço base de alojamento ; longitude ; latitude ; data de início de época com preço diferente ; data de fim de época ; percentagem de alteração de preço”, sendo que os últimos 3 elementos são opcionais e podem ser repetidos. Exemplo: “1;Porto;30;-8.610243;41.149968;1/12;28/2;0.8;1/6;1/7;1.0;1/7;1/9;1.3”.

O algoritmo

Tendo adotado o algoritmo de Dijkstra para a resolução do primeiro problema, chegamos à conclusão que não basta implementar a versão mais ingénua deste, obrigando-nos a modificá-lo de acordo com as nossas necessidades.

De modo a tornar a procura do caminho ótimo mais direta, e sob a premissa que caminhos mais curtos serão mais baratos (com base em várias limitações, tal como combustível), efetuamos primeiro uma pesquisa sobre todos os vértices tentando encontrar os caminhos mais curtos para o

nó de destino. Assim, e também porque o grafo em questão não é acíclico, conseguimos poupar bastante tempo de computação no algoritmo de Dijkstra, onde as operações são mais caras em termos computacionais, fazendo um pré-processamento do grafo: utilizamos um algoritmo para calcular a distância mínima de todos os nós ao nó que corresponde ao nosso destino, ficando esta informação guardada em cada nó. Quando executarmos o algoritmo, impomos a regra aquando da seleção da aresta de menor peso que o nó de destino não deve ter uma distância igual ou superior ao nó atual. Caso não consigamos encontrar uma aresta que respeite esta regra, podemos assumir que não há ligação possível entre os dois nós escolhidos.

Adicionalmente, temos o problema do custo dos alojamentos. A variação do custo dos alojamentos com os dias e o facto de diferentes viagens (seja entre nós diferentes, seja entre diferentes meios de transporte para o mesmo nó) demorarem diferentes períodos de tempo faz com que haja demasiadas possibilidades para serem testadas num espaço de tempo minimamente razoável. Assim, escolheremos uma solução mais simples, que não sendo sempre a ótima, será na maior parte dos casos razoavelmente boa. Esta solução consiste em somar ao peso de cada aresta (custo, em função do meio de transporte) o custo do alojamento no nó de destino, em função da data de partida mais o tempo da viagem considerada; esta soma só ocorre durante a execução do algoritmo, mais precisamente durante a comparação dos pesos, pois os valores variam sempre com a data.

De modo a facilitar a descrição da complexidade temporal e espacial do algoritmo, daqui em diante utilizaremos a seguinte convenção:

- V representa o número de nós do grafo.
- E representa o número de arestas de todos os nós do grafo (V).
- E(V) representa o número de arestas de cada nó do grafo.

Apesar de o grafo ser bastante denso, vamos admitir que não temos o caso extremo onde o número de arestas total do grafo é dado por $E = V^2$; ao mesmo tempo, temos que estabelecer um limite inferior. Nesse caso, admitindo que todos os nós estão conectados, e que para cada viagem numa direção existe uma viagem na direção oposta, esse limite pode ser dado por $E = 2 \bullet (V - 1)$. Efetuando a soma parcial destes dois limites, obtemos $E = (V^2 + 2 \bullet (V - 1)) / 2$, que pode ser simplificado para $E = V^2 / 2 + V - 1$ que por sua vez, em notação *Big O*, é equivalente a $E = V^2$, e que implica que $E(V) = V$. Verificamos que, mesmo não considerando o caso extremo, para os casos médios $E = V^2$ é a expressão que mais se aproxima da nossa situação.

Podemos assim descrever o algoritmo pelo seguinte código (em C++):

```
// ===== Pre-processing =====

for (auto v_it = vertexes.begin(); v_it != vertexes.end(); ++v_it)
{
    v_it->distance = INT_MAX;
}
int currentDistance = dest->distance = 0;

bool processing;
do
{
    processing = false;

    for (auto v_it = vertexes.begin(); v_it != vertexes.end(); ++v_it)
    {
        for (auto e_it = v_it->edges.begin(); e_it != v_it->edges.end(); ++e_it)
        {
            Vertex * destination = e_it->destination;

            if (destination->distance == currentDistance &&
                v_it->distance > currentDistance + 1)
            {
                processing = true;

                v_it->distance = currentDistance + 1;
            }
        }
    }

    ++currentDistance;
}
while (processing);

```

Esta parte do código é referente ao pré-processamento feito para tornar a pesquisa posterior mais rápida. Será potencialmente a parte mais exaustiva computacionalmente, uma vez que consiste em três *loops* aninhados, embora as operações que realize sejam de baixo custo computacional.

No entanto, e por uma questão de argumento, consideraremos aqui o caso mais extremo, com complexidade temporal:

$$O(V^2 \bullet E(V))$$

e que, tendo em conta as considerações anteriores a respeito do grafo, é equivalente a:

$$O(V^3)$$

Porém, o nível de complexidade espacial é sempre:

$$O(1)$$

pois não necessita de guardar informação temporária relativa a nós ou arestas.

```
34      // ===== Custom Dijkstra Algorithm =====
35
36      for (auto v_it = vertexes.begin(); v_it != vertexes.end(); ++v_it)
37      {
38          v_it->totalPrice = INT_MAX;
39          v_it->path = nullptr;
40          v_it->processing = false;
41      }
42      src->totalPrice = 0;
43
44      // Requires overload of operator>(Vertex * v1, Vertex * v2)
45      priority_queue<Vertex *, vector<Vertex *>, std::greater<Vertex *>> p_queue;
46      p_queue.push(src);
47
48      while (!pqueue.empty())
49      {
50          Vertex * from = p_queue.top();
51          p_queue.pop();
52
53          bool possible = false;
54
55          for (auto e_it = from->edges.begin(); e_it != from->edges.end(); ++e_it)
56          {
57              Vertex * to = e_it->destination;
58
59              if (to->distance < from->distance &&
60                  from->totalPrice + e_it->tripPrice + to->getLodgingPrice(date + e_it->getTripDuration()) < to->totalPrice)
61              {
62                  possible = true;
63
64                  date += e_it->getTripDuration();
65
66                  to->totalPrice = from->totalPrice + e_it->tripPrice + to->getLodgingPrice(date);
67                  to->path = from;
68
69                  if (!to->processing)
70                  {
71                      to->processing = true;
72
73                      p_queue.push(to);
74                  }
75              }
76          }
77
78          if (!possible)
79          {
80              return vector<Vertex *>(); // Failed to find path, return empty vector
81          }
82      }
83
```

Esta segunda parte do código já é referente ao algoritmo de Dijkstra modificado. O primeiro *loop* (**for**) corre um número de vezes igual a V. O segundo *loop* (**while**) corre um número de vezes também aproximadamente igual a V (uma vez que o grafo a considerar é completamente ligado, ou seja, não há nós sem ligações, será exatamente igual a V). Já o terceiro *loop* (**for**), dentro do segundo, correrá, para cada iteração deste, um número de vezes igual a E(V).

Assim, temos complexidade temporal:

$$O(V + V \bullet E(V))$$

que é equivalente a:

$$O(V^2)$$

Já a nível de complexidade espacial, uma vez que o algoritmo necessita de guardar informação relativa aos nós, ocupa no pior caso:

$$O(V)$$

```
84      // ===== Process and return calculated path =====
85
86      stack<Vertex *> tmp_path;
87      Vertex * tmp = dest;
88      while (tmp_path.push(tmp), tmp != src)
89      {
90          tmp = tmp->path;
91      }
92
93      vector<Vertex *> path;
94      while (!tmp_path.empty())
95      {
96          path.push_back(tmp_path.top());
97          tmp_path.pop();
98      }
99
100     return path;
```

Esta ultima parte relaciona-se com o output do resultado obtido. Tanto o primeiro loop como o segundo correm um número de vezes no máximo igual a V.

Então, a complexidade temporal pode ser dada por:

$$O(V + V)$$

que é equivalente a:

$$O(V)$$

Também a complexidade espacial, devido ao facto de o algoritmo criar dois *containers* com tamanho no máximo igual a V, pode ser obtida por:

$$O(V + V)$$

que, como já observado, é equivalente a:

$$O(V)$$

==========

Atendendo a todas estas considerações, a complexidade temporal de todo o algoritmo é:

$$O(V^3 + V^2 + V)$$

e uma vez que esta é obtida pelo termo de maior crescimento, a função que melhor a aproxima é:

$$O(V^3)$$

Enquanto que a nível de complexidade temporal do algoritmo, temos:

$$O(1 + V + V)$$

pelo que, seguindo pelo mesmo raciocínio, a função que melhor a aproxima é:

$$O(V)$$

Adaptando o algoritmo para múltiplos destinos

Relativamente ao segundo problema, a maneira mais simples, mas não necessariamente ótima, de chegar à solução, é correr o algoritmo de Dijkstra modificado partindo do ponto inicial para cada um dos destinos escolhidos pelo utilizador, escolhendo a opção mais barata e definindo o destino como novo ponto inicial, executando os passos anteriores novamente para cada um dos destinos restantes, e repetindo este processo até que esgotemos a lista de destinos definida pelo utilizador. Em termos computacionais, isto significa que correremos o algoritmo $(D^2 + D) / 2$ vezes, onde D é o número de destinos escolhidos pelo utilizador, pelo que a complexidade temporal, no pior caso, se aproxima de:

$$O(D^2 \bullet V^3)$$

Com efeito, não é uma solução muito boa em termos de custos computacionais, mas veremos mais à frente que com efeito, o custo será muito mais baixo, devido a certas assunções que fizemos.

Além disto, há que considerar ainda que em termos de complexidade espacial, temos:

$$O(D + V)$$

pois precisamos de guardar informação relativa ao custo de cada vez que se corre o algoritmo da secção anterior, que no caso extremo (que ocorre da primeira vez que o algoritmo adaptado é executado) ocupa espaço igual a D.

Uma das vantagens deste método é que no entanto não correremos o risco de repetirmos viagens a sítios onde já estivemos, uma vez que, para chegar a um dos destinos selecionados, se tiver que passar por outro dos selecionados, então este último será obrigatoriamente o destino mais barato (pelo menos mais barato que o primeiro), uma vez que requer menos viagens.

Algoritmos adicionais implementados:

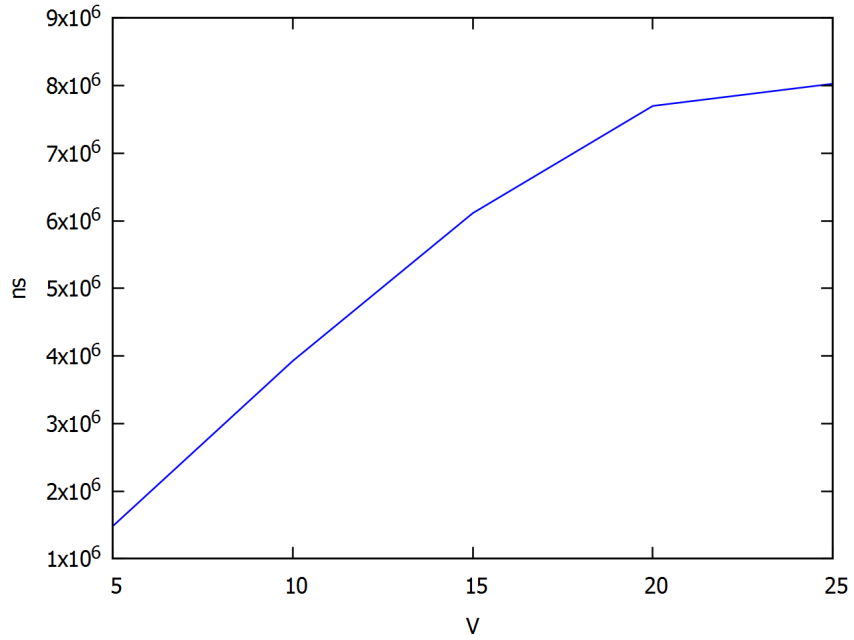
Variação do algoritmo de Gauss para a determinação do dia da semana:

$W = (d + \mathit{floor}(2.6 \bullet m - 0.2) + y + y / 4 + c / 4 - 2 \bullet s) \% 7$, onde:

- d é o dia da semana;
- m é o número do mês – 2, exceto janeiro = 11 e fevereiro = 12;
- s é o século;
- y é são os 2 últimos algarismos do ano;
- W é o dia da semana em que domingo = 0 e sábado = 6.

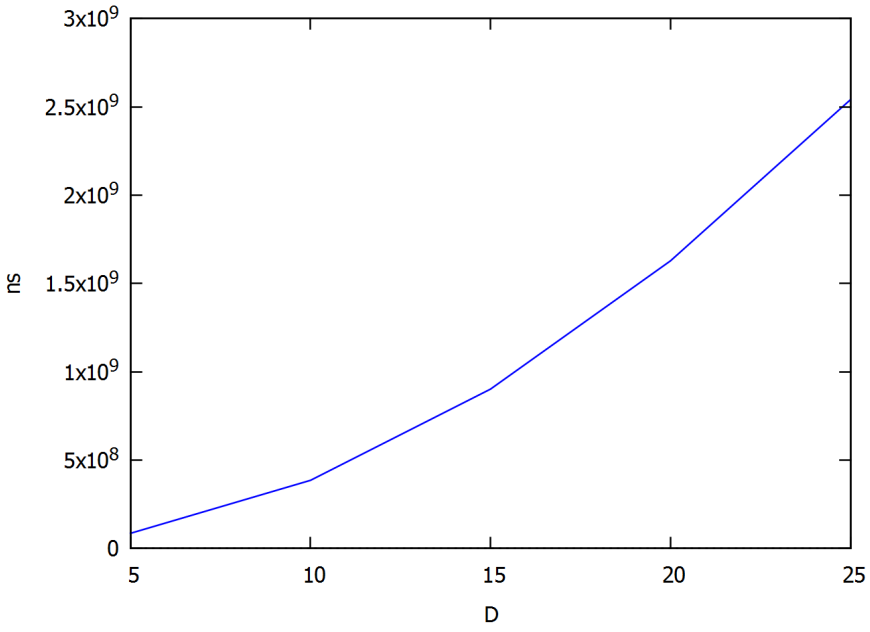
Breve análise da complexidade temporal

Tendo corrido os nossos algoritmos sobre alguns dados, conseguimos comprovar algumas das observações que fizemos na análise anterior. Para cada valor no eixo horizontal dos gráficos a seguir apresentados, corremos os algoritmos 100 vezes. Para o primeiro algoritmo, para uma pequena amostra de nós:

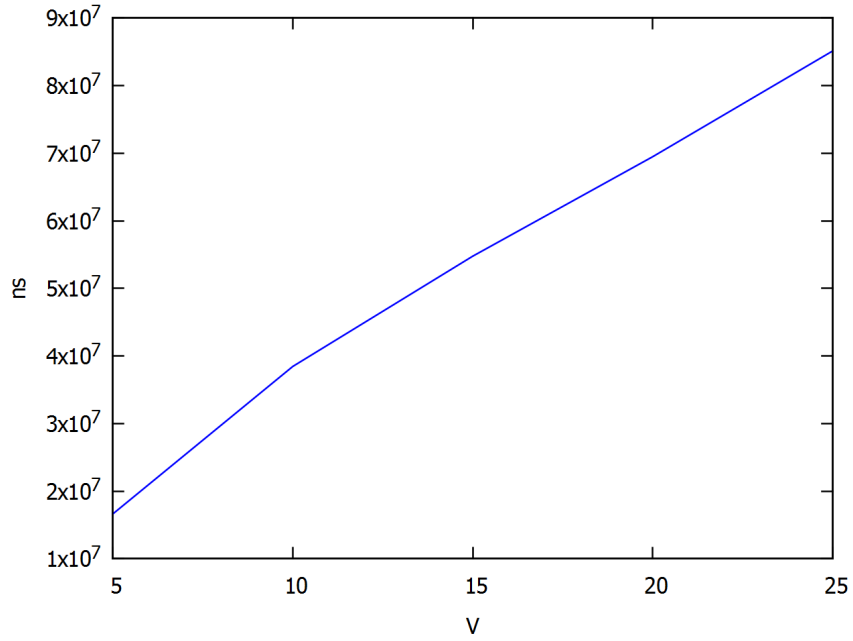


Os valores do tempo sobem substancialmente com a inserção de apenas alguns nós (relembramos que as expressões foram completamente exprimidas através de V), mas com efeito, a função aproxima-se mais de uma função logarítmica do que de uma polinomial, como foi inicialmente calculado. Acreditamos que tal ocorre pois como observamos anteriormente, as operações de maior custo computacional ocorrem no algoritmo de Dijkstra, e não no pré-processamento, de onde obtivemos a expressão original. Por outro lado, nós assumimos para o algoritmo de Dijkstra um pior caso de $O(V^2)$ que provinha de um número de arestas $E = V^2$, porém esse caso não ocorre: o número de arestas sobre o qual testamos não é assim tão alto e, como já foi comprovado por testes de outros autores, no caso médio, a complexidade temporal do algoritmo de Dijkstra, quando implementado com uma *priority queue*, é dada por $O(E + V \bullet \log V)$, o que se correlaciona com os valores observados.

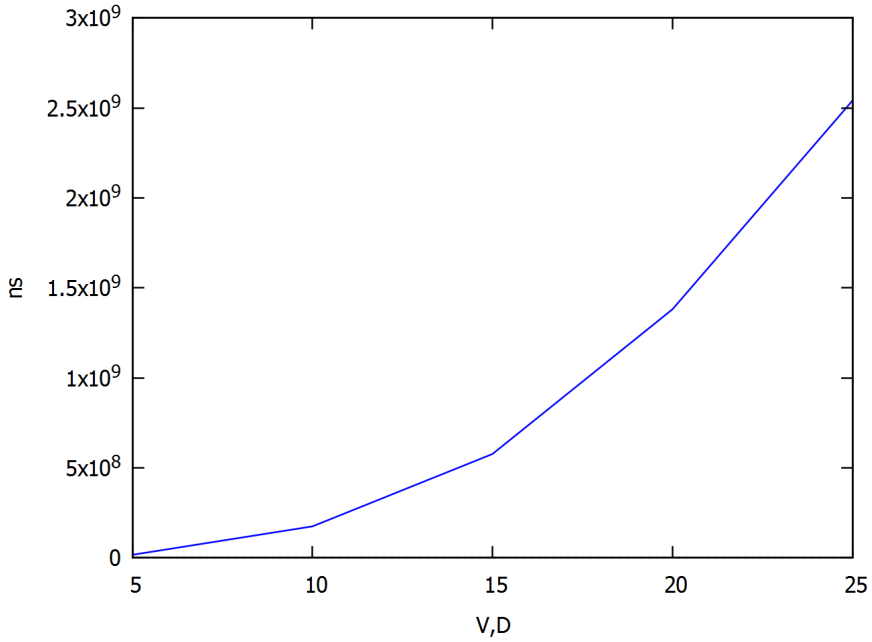
Já para o segundo algoritmo, obtivemos os seguintes resultados, num grafo com 25 nós:



Estes valores foram obtidos com diferentes valores de D, e verificamos que se aproximaram mais da assimptota obtida pela expressão $O(D^2 \bullet V^3)$. Adicionalmente, com o mesmo algoritmo corrido em vários grafos com variação no número de nós, e utilizando o mesmo input para cada grafo de D = 5, obtivemos os seguintes valores:



O comportamento levemente logarítmico comprova as observações feitas relativamente ao primeiro gráfico. Quando comparamos este gráfico com o gráfico obtido para o algoritmo corrido nos mesmo grafos, mas desta vez com um input D = V:



chegamos facilmente à conclusão de que o fator D é muito mais significativo do que V no que toca à variação da complexidade temporal.

Principais Dificuldades

As principais dificuldades na elaboração do trabalho solicitado consistiram na estruturação do programa para depois nos facilitar a utilização dos conceitos dum grafo.

Tivemos também alguns problemas com a utilização do grafo criado durante as aulas práticas tendo o trabalho final um grafo personalizado por nós, mas semelhante ao dito cujo.

Mesmo assim, estamos convictos de que realizamos com acerto a proposta apresentada.

Referências bibliográficas

1. Cheriton School of Computer Science - University of Waterloo, 1998, “How to determine the day of the week, given the month, day and year”. Disponível em: https://cs.uwaterloo.ca/~alopez-o/math-faq/node73.html. Data de acesso: 1 de março de 2017.
2. Wikipedia, 2017, “Determination of the day of the week”. Disponível em: https://en.wikipedia.org/wiki/Determination_of_the_day_of_the_week. Data de acesso: 1 de março de 2017.
3. Wikipedia, 2017, “Dijkstra’s Algorithm”. Disponível em: https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm. Data de acesso: 10 de março de 2017.