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, 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, um vetor de monumentos que aí existem 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.

Num segundo objetivo, visamos a implementação de alguns algoritmos simples para correspondência de *strings*, que estenderão a funcionalidade do programa para permitir ao utilizador a escolha de um destino com base num monumento que queira visitar.

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, como ocorrerá 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:

- Calcular a viagem mais económica para um destino.
- Calcular a viagem mais económica para vários destinos.
- Encontrar cidade onde se situa o monumento procurado.
- Visualizar grafo.

Dados de entrada:

- Cidade em que se pretende iniciar a viagem;
- Nome (exato ou aproximado) do monumento que se pretende visitar;

- 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*(classe que representa os alojamentos), *monuments*(vetor de nomes dos monumentos) 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”.

Na linha seguinte, é necessário uma flag “#Monuments_start” que indica que as próximas linhas representam os nomes dos monumentos existentes nessa cidade, sendo essa secção terminada pela flag “#Monuments_end”.

Exemplo: “#Monuments_start\nTorre dos Clérigos\n#Monuments_end” (onde ‘\n’ representa uma nova linha).

Os algoritmos

Este relatório focar-se-à principalmente nos algoritmos respeitantes ao segundo objetivo do trabalho, tanto na análise teórica como na empírica; uma análise aos algoritmos utilizados para a resolução do primeiro objetivo já foi feita num relatório anterior.

Procura de destino por nomes de monumentos:

Existe a opção de pesquisa de um monumento pelo nome, pesquisa essa que devolverá o nome da cidade onde se situa esse monumento. Cada nó do grafo contém um vetor composto pelos nomes de monumentos que existam nessa cidade, e é utilizado um algoritmo de pesquisa de *strings* que compara esses nomes com o *input* do utilizador para determinar os potenciais destinos onde o monumento exista. Dizemos potenciais pois o algoritmo aceita *input* que possa ser pouco exato (erros de escrita, devido ou não a desconhecimento da parte do utilizador), tentando aproximá-lo aos nomes e devolvendo aqueles que mais se assemelham ao *input*. Portanto dois algoritmos são utilizados nesta pesquisa, o primeiro realizando uma pesquisa exata, procurando nomes onde o *input* do utilizador esteja contido; o segundo fazendo uma pesquisa aproximada, devolvendo nomes até um certo nível de proximidade com o *input*. Para o primeiro, utilizamos uma implementação simples do algoritmo de Knuth-Morris-Pratt (KMP), que executa com complexidade temporal O(N) (onde N é o número de caracteres do nome), como já veremos a seguir. Para o segundo, utilizamos um algoritmo com base na *edit distance*[1] das duas *strings* comparadas (nome e *input*), verificando se o valor devolvido por esse algoritmo está dentro de um certo limiar, e fornecendo esse nome como hipótese de destino ao utilizador caso esteja.

O seguinte código descreve o algoritmo KMP (em C++):

```cpp
// ===== Pre-compute table =====

size_t m = pattern.length();
vector<int> prefix(m);

prefix[0] = -1;

for (int k = -1, q = 1; q < m; ++q)
{
    while (k > -1 && pattern[k + 1] != pattern[q])
    {
        k = prefix[k];
    }
    if (pattern[k + 1] == pattern[q])
    {
        ++k;
    }
    prefix[q] = k;
}

```

Nesta primeira parte do código, a tabela de correspondência parcial é computada, que será posteriormente utilizada pelo algoritmo em si. Apesar de ter dois *loops* aninhados, o *loop* interior por norma não terá muitas iterações, uma vez que é necessária uma elevada repetição de letras e padrões para que isso aconteça.

Assim, podemos limitar a complexidade temporal pela expressão:

$$O(M)$$

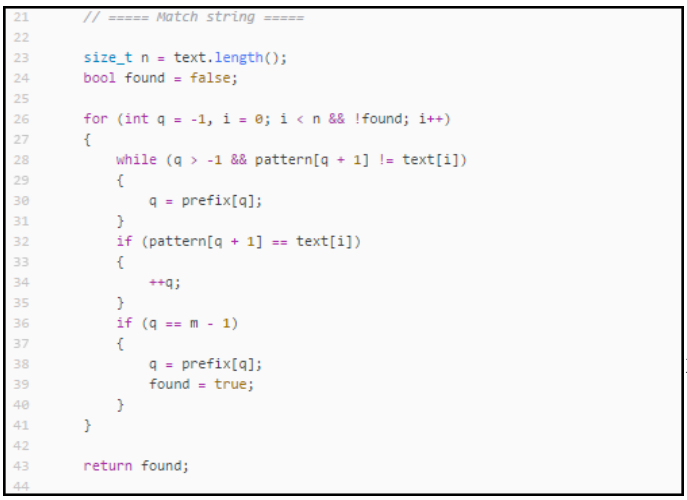
onde M é o número de caracteres no *input* do utilizador.

Além disso, devido à necessidade de criação de um vetor com o mesmo número de elementos que o número de caracteres no *input*, o nível de complexidade espacial é também da ordem:

$$O(M)$$

---

[1]: termo utilizado para quantificar as diferenças entre duas cadeias de caracteres através do número de operações que seriam necessárias (inserção, substituição e remoção de caracteres) para transformar uma das cadeias na outra.



```
21      // ===== Match string =====
22
23      size_t n = text.length();
24      bool found = false;
25
26      for (int q = -1, i = 0; i < n && !found; i++)
27      {
28          while (q > -1 && pattern[q + 1] != text[i])
29          {
30              q = prefix[q];
31          }
32          if (pattern[q + 1] == text[i])
33          {
34              ++q;
35          }
36          if (q == m - 1)
37          {
38              q = prefix[q];
39              found = true;
40          }
41      }
42
43      return found;
44
```

Esta segunda parte do código já se refere ao algoritmo KMP em si, e faz uso da tabela anterior para o cálculo. De modo semelhante à parte anterior, tem dois *loops* aninhados, porém o *loop* interior não é mais que uma verificação de igualdade entre caracteres, pelo que as iterações dependem diretamente do número de repetições e padrões de caracteres tanto no nome ("*text*") como no *input* ("*pattern*"). O *loop* exterior faz incrementos constantes em "i", pelo que o número de iterações em diferentes execuções do algoritmo para o mesmo nome não irá variar.

Mais uma vez, podemos limitar a complexidade temporal por:

$$O(N)$$

No entanto, podemos considerar a complexidade espacial desta secção de código:

$$O(1)$$

dado que apenas o vetor criado na secção de código anterior é utilizado, não sendo criado mais nenhum *container*.

==========

Com tudo isto em consideração, podemos aproximar a função da complexidade temporal do algoritmo pela expressão:

$$O(M + N)$$

e tendo em conta que o termo N varia drasticamente dependendo do número de caracteres de cada *string* a que o *input* é comparado, expressaremos a função por:

$$O(M)$$

Em termos de complexidade espacial, uma vez que apenas um vetor com o tamanho do *input* é criado durante a execução do algoritmo, esta pode ser dada exatamente por:

$$O(M)$$

Pesquisa aproximada do nome de um monumento:

Como descrito na secção anterior, caso a pesquisa pelo algoritmo de KMP não dê frutos, é corrido um algoritmo adicional, que procura nomes com apenas uma aproximação ao *input* dado (sendo especialmente útil no caso de engano ou desconhecimento na escrita do nome por parte do utilizador).

```
size_t m = pattern.length();
size_t n = text.length();
vector<int> d(n + 1);

int old, neww;

for (int j = 0; j <= n; j++)
{
    d[j] = j;
}

for (int i = 1; i <= m; i++)
{
    old = d[0];
    d[0] = i;

    for (int j = 1; j <= n; j++) {
    {
        if (pattern[i - 1] == text[j - 1])
        {
            neww = old;
        }
        else
        {
            neww = min(old, d[j]);
            neww = min(neww, d[j - 1]);

            ++neww;
        }

        old = d[j];
        d[j] = neww;
    }
}

return d[n];
```

O algoritmo contém três *loops*, estando dois deles aninhados, e uma vez que cada um desses percorre cada uma das *strings* fornecidas (o nome e o *input*), a complexidade temporal será potencialmente quadrática (se as *strings* forem de tamanhos idênticos). Todavia, as operações que realiza dentro destes *loops* são de baixo custo computacional, pelo que o tempo efetivamente gasto não será muito elevado, mesmo com um aumento do número de nomes a comparar (caso a base de dados cresça).

Portanto, a complexidade temporal é dada por:

$$O(N \bullet M)$$

Adicionalmente, há também a criação de um vetor com tamanho igual ao do nome do monumento, tornando a complexidade espacial:

$$O(N)$$

==========

Ocasionalmente, tanto o algoritmo KMP como este que acabou de ser descrito irão ser executados (este último sendo executado sempre que o de KMP falhar). Nesta situação, a complexidade temporal de toda a execução pode ser dada por:

$$O(N + N \bullet M)$$

que se reduz a:

$$O(N \bullet M)$$

Adicionalmente, temos complexidade espacial linear para ambos os algoritmos, no entanto, uma vez que estes não correm ao mesmo tempo, e os *containers* não têm que permanecer após a sua execução, podemos obter a função da complexidade espacial total pelo maximizante das duas funções anteriores. Com grande probabilidade, este máximo será na maior parte dos casos o nome do monumento, pelo que podemos considerar a expressão:

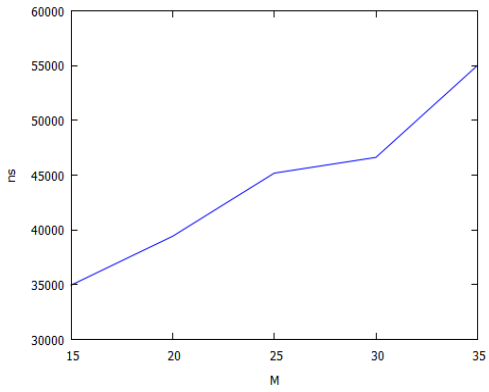
$$O(N)$$

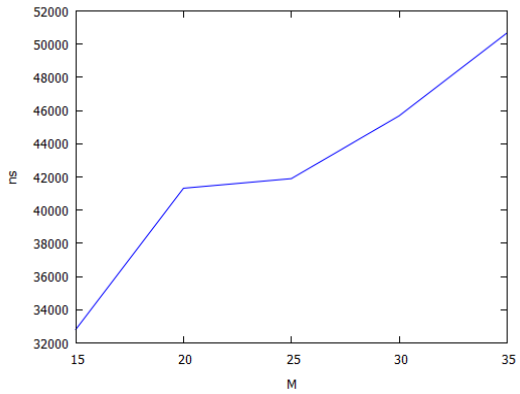
como aquela que melhor aproxima a complexidade espacial da execução.

Breve análise da complexidade temporal

Tendo corrido os nossos algoritmos sobre alguns dados, conseguimos comprovar algumas das observações que fizemos na análise anterior. Estes testes foram feitos num computador com um processador Intel® Core i7-4790K e com 8 GB de RAM disponível. Não foi feita nenhuma otimização sobre o código por parte do compilador.

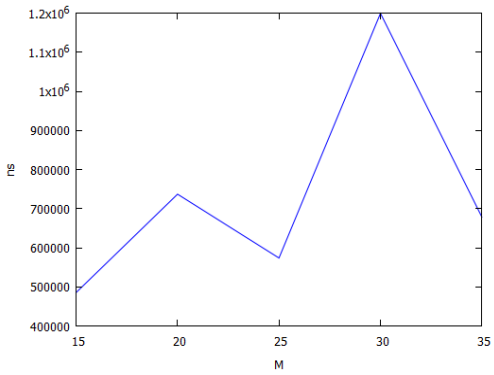
Para o KMP, temos os seguintes resultados, de dois testes separados, para *inputs* com diferente número de caracteres:

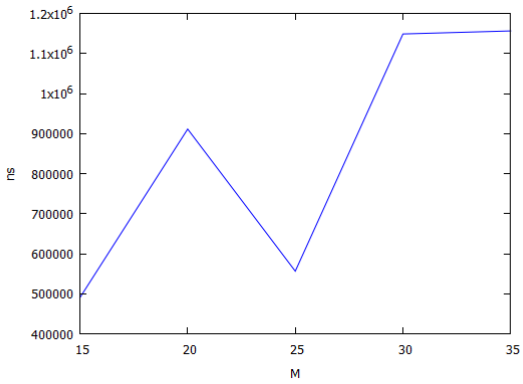




Verificamos que, como calculado, o algoritmo mantém um comportamento algo linear, ainda que demonstrando algumas irregularidades, potencialmente consequência das variações nos padrões das *strings*.

Já para o algoritmo da aproximação de *strings*, obtivemos os seguintes valores:





que revelam grandes irregularidades para uma variação baixa de M, mas que, todavia, mantêm-se entre limites de tempo bem delineados (entre 0.5 e 1.2 milissegundos). Apesar de existir um crescimento em y claro, caso tivéssemos variado o número de caracteres nos nomes dos monumentos mais drasticamente, e relembrando que a complexidade temporal dada para este algoritmo é de O(N • M), provavelmente veríamos uma aproximação do gráfico a uma função mais linear.

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.
4. University of Chile – 2000 “A Guided Tour to Approximate String Matching ”.
   Disponível em: http://users.csc.calpoly.edu/~dekhtyar/570-Fall2011/papers/navarro-approximate.pdf.
   Data de acesso: 21 de maio de 2017.