Trabalho Prático 2: Reducing the costs!

1. Introdução

1.1. Descrição do problema

Nesse trabalho temos um problema de logística a ser resolvido. Um caminhão de entrega de produtos de uma empresa deve percorrer um trajeto por várias cidades, pegando produtos em algumas dessas cidades e deixando-os em outras, e voltando à cidade de origem em seguida. Deve-se garantir que esse trajeto será feito de forma ótima, ou seja, gastando a menor quantidade de combustível possível, o que nesse caso é o mesmo que garantir que o caminhão percorra a menor distância possível.

O caminhão deve percorrer todas as cidades, e cada cidade deve ser visitada apenas uma vez. Além disso, o programa receberá algumas restrições, que indicam que determinada cidade deve ser visitada antes de outra, ou seja, existem produtos em determinada cidade que devem ser entregues em outra.

Esse problema é NP-Completo, como provaremos a seguir. Veremos mais à frente que esse problema tem complexidade fatorial, o que faz com que o programa leve um tempo enorme para executar até mesmo em entradas consideradas não tão grandes. Para viabilizar a execução do programa, executaremos "podas", ou seja, utilizaremos de mecanismos que evitem buscas desnecessárias quando nos deparamos com uma busca que claramente não trará um resultado ótimo ou permitido.

1.2. Descrição teórica e classificação do problema

- **Formalização do problema como um problema de decisão**

  O problema dado pode ser tratado como um problema de decisão na seguinte forma:

  Dado um grafo completo G, um número real positivo k, e um conjunto de restrições de precedência de vértices R, é possível gerar um caminho que visite todos os vértices de G exatamente uma vez, volte à origem, respeite as restrições impostas por R, e cuja soma dos pesos de suas arestas seja menor ou igual a k?

- **Prova de que o problema de decisão é NP-Completo**

  Primeiramente provaremos que o problema pertence à classe NP. Para isso, podemos criar um algoritmo determinista que confira se uma solução para esse problema é válida em tempo polinomial. Segue o algoritmo:

```
confere(vértices V, arestas E, caminho C, restricoes R, int k):
    atual = C.inicio
    proximo = atual.proximo
    soma =0
    visitado = {}
    enquanto proximo != origem
        visitado[atual] = true
        para cada restricao r em R:
            se !visitado[r.requisito] e r.destino = proximo
                return false
        se soma + E(atual, proximo) > k
            return false
        atual = proximo
        proximo = proximo.proximo
    return true
```

Esse algoritmo passa por todos os vértices, e para cada vértice ele checa todas as restrições. Portanto, tem complexidade polinomial dada por O(|V| * |R|).

- **Mostrando a existência de uma instância fácil para o problema de decisão**

  Podemos de forma simples mostrar duas instâncias fáceis para esse problema:

  Instância 1: Dado um grafo G cujo número de vértices |V| = 2, esse grafo terá apenas um caminho possível, que consiste em sair da origem para a única cidade restante e voltar. Não faz sentido que haja uma restrição nesse caso. Esse caminho terá o tamanho igual a duas vezes o peso da única aresta existente. Basta comparar esse valor com k.

  Instância 2: Dado um grafo G cujo número de vértices |V| = 3, esse grafo terá três arestas, que formarão um único ciclo, ou seja, um único caminho. Haverá deadlock se houver uma restrição que bloqueie as duas direções de uma mesma aresta (visitar A antes de B e B antes de A). O caminho terá tamanho igual a duas vezes a soma dos pesos das arestas. Basta comparar esse valor com k.

- **Descrição das podas e análise de eficiência de cada poda**

  O programa desenvolvido possui duas podas:

  - **Restrições**: Como temos determinadas restrições de quais cidades devem vir antes de outras, podemos usar isso a nosso favor. Se temos uma restrição que diz que A deve vir antes de B, podemos abortar um caminho assim que encontramos a cidade B, se sabemos que a cidade A não foi visitada ainda. Assim, evitamos percorrer todos os caminhos que continuaríam a partir desse ponto e que já estariam certamente inválidos. Essa poda não se aplica em qualquer situação. O caso em que a poda se torna mais eficaz ocorre quando

as restrições delimitam um único caminho, por exemplo, tendo cinco cidades: A, B, C, D e E, se as restrições forem A -> B, B -> C, C -> D, D -> E, (sendo que X -> Y representa que X deve vir antes de Y) temos apenas um caminho válido, e apenas esse caminho será checado. Por outro lado, o caso em que essa poda será menos eficaz será quando não houverem restrições. Quando isso ocorrer, não haverá forma alguma de determinar que um caminho é necessariamente inválido, e portanto teremos que percorrer todos os caminhos possíveis.

- **Menor caminho**: Se estamos em determinado ponto no programa no qual já encontramos um caminho que percorre todas as cidades e volta para a origem, e estamos seguindo um outro caminho possível entre as cidades, podemos chegar em um ponto no qual a distância percorrida até a cidade atual já ultrapassa o tamanho do menor caminho que encontramos. Nesse caso, todos os caminhos que continuam a partir desse local terão tamanho maior que o menor tamanho já encontrado, e não precisamos nem checá-los. Portanto, abortamos a procura nesse caminho, voltando para a cidade anterior. O caso em que essa poda é mais eficiente ocorre quando existe um ciclo que resolve o problema e que possui tamanho x, e todas as arestas que não pertencem a esse ciclo tem tamanho maior que x, e então visitamos o ciclo correto no primeiro caminhamento. Assim, no momento em que encontrarmos esse ciclo, todos os outros caminhos serão descartados. O pior caso ocorre quando as arestas ligadas na origem são as que possuem maior peso e as que não estão ligadas na origem possuem pesos muito pequenos se comparados aos que se ligam na origem, pois nesse caso, muitas vezes veremos que já temos um caminho melhor que o atual apenas quando estivermos voltando para a origem, e por isso basicamente teremos que percorrer todos os caminhos existentes.

2. Solução do problema

Para cada caso de teste, primeiramente lemos todas as cidades e restrições impostas. Enquanto lemos as restrições, conferimos se há algum ciclo óbvio nelas (isto é, a cidade a deve vir antes da b, e a b antes da a). Caso haja, reportamos o deadlock e vamos para o próximo caso. Caso não haja deadlock nessa etapa, seguimos com o procedimento. Criamos um grafo completo e não-direcionado com as cidades, onde cada aresta possui como peso a distância entre as cidades que está ligando. Em seguida, chamamos a função que encontra o trajeto de menor distância percorrida que atende aos requisitos. Caso haja deadlock, essa função retornará um valor infinito (que representamos, no programa, como o maior número de tipo double possível, DBL_MAX), do contrário, retornará a menor distância percorrida. Avisamos que houve deadlock, se for o caso. Do contrário, exibimos o resultado obtido e vamos para o próximo caso de teste.

A função que encontra o menor trajeto funciona conforme o seguinte pseudo-código, onde C é cidade que estamos averiguando, deslocamento é o quanto andamos para chegar na cidade C considerando o caminho utilizado atualmente, e menorCaminho é o tamanho do menor caminho que encontramos até agora que resolve o problema:

```
visitaCidade (C, deslocamento):
    se cidades[C].visitada e C == origem e todasCidadesVisitadas()
        menorCaminho = deslocamento
    do contrário, se cidades[C].visitada
        return
    cidades[C].visitada = true
    para cada cidade D diferente de C
        se distância(C, D) + deslocamento < menorCaminho e restricoesAtendidas(D)
            visita(D, distância(C, D) + deslocamento)
    cidades[C].visitada = false
```

Por fim, teremos o tamanho do menor caminho possível na variável menorCaminho. A primeira chamada dessa função deve ser feita passando como argumentos a cidade de origem e deslocamento 0.

3. Análise de complexidade

A análise de complexidade será feita baseada na variável n, que representa a quantidade de cidades inserida no programa. As funções mais importantes são analisadas a seguir:

- **constroiGrafoCidades**: Recebe as cidades e constrói um grafo não-direcionado completo a partir delas. Essa função passa por cada aresta necessária no grafo. Como sabemos, um grafo completo possui n*(n-1)/2 arestas, portanto essa é a complexidade da função. Efetuando a multiplicação, teremos (n^2-n)/2 operações, portanto, essa função é **O(n^2)**.
- **geraCaminhoMinimo**: Função que encontra o menor caminho válido.. Essa função basicamente chama a função **visita** passando para ela o vértice de origem. A função visita chama a si mesma recursivamente passando como argumento cada um dos vértices adjacentes ao vértice atual que ainda podem ser visitados, ou seja, ainda não foram visitados no caminho atual e cujas restrições tenham sido atendidas. Quando não houver mais nenhum vértice possível, volta à origem, e marca a distância percorrida no caminho atual como a menor distância, caso realmente seja. Ou seja, quando a função visita é chamada no vértice de origem, chamará a si mesma n-1 vezes. Para cada uma das n-1 chamadas a visita, ela será chamada n-2 vezes, e assim por diante, até zero. Teremos uma complexidade n * (n – 1) * ... * 1 = n!. Poranto, teremos complexidade fatorial, **O(n!)**.
- **main**: A função main chama a função **constroiGrafoCidades** e **geraCaminhoMinimo** uma vez para cada caso de teste. Sendo T o número de casos de teste, teremos complexidade T * (n^2 + n!), ou seja, **O(T * n!)**.

4. Análise experimental

Para a análise experimental do código, utilizamos arquivos de testes com quantidades variadas de cidades. Não utilizamos restrições, assim podemos ver o comportamento das funções sem essa interferência. Fizemos isso porque medir experimentalmente o efeito causado pelas restrições é algo extremamente complicado, já que não podemos utilizar a quantidade de restrições como parâmetro para essa análise, isto é, uma quantidade pequena de restrições feitas de determinada maneira pode diminuir drasticamente o tempo de execução, enquanto uma quantidade grande de restrições feitas de outra forma pode acabar por não diminuir o tempo tomado de forma perceptível. Pelo mesmo motivo, também retiramos a poda onde o programa não verifica um caminho se ele já possui uma distância maior que a menor encontrada.

Para medir o tempo passado, utilizamos a biblioteca <time.h> para contar o número de clocks necessários para executar cada caso de teste, e usamos a variável CLOCKS_PER_SEC para calcular o tempo tomado por cada caso. Os testes foram feitos em uma máquina com processador Core i5 com frequência de 2.6GHz e 4GB de memória RAM.

Medimos o tempo levado para executar o programa com a quantidade de cidades variando entre 2 e 13. Para quantidades superiores a 13, o tempo superaria muito o limite do gráfico (seria possível fazer o gráfico, mas a visualização dos dados ficaria comprometida). Executamos cada caso de teste cinco vezes, e o tempo final foi a média de todos os tempos obtidos.

Obtivemos o seguinte gráfico. O eixo horizontal do gráfico representa a quantidade de cidades recebidas, e o eixo vertical representa o número de segundos tomados para a execução do programa.

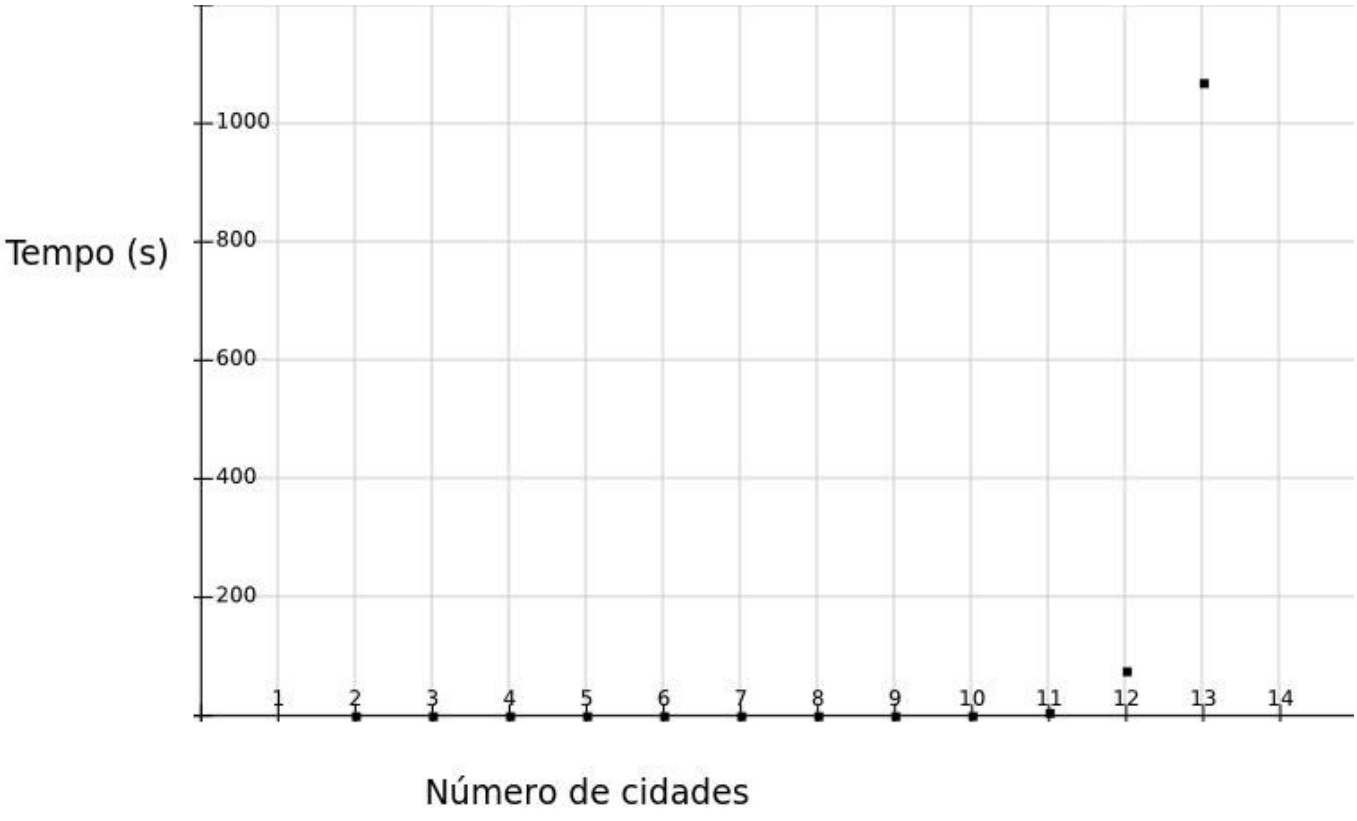

Gráfico 1: Número de cidades x Tempo de execução com todos os pontos medidos

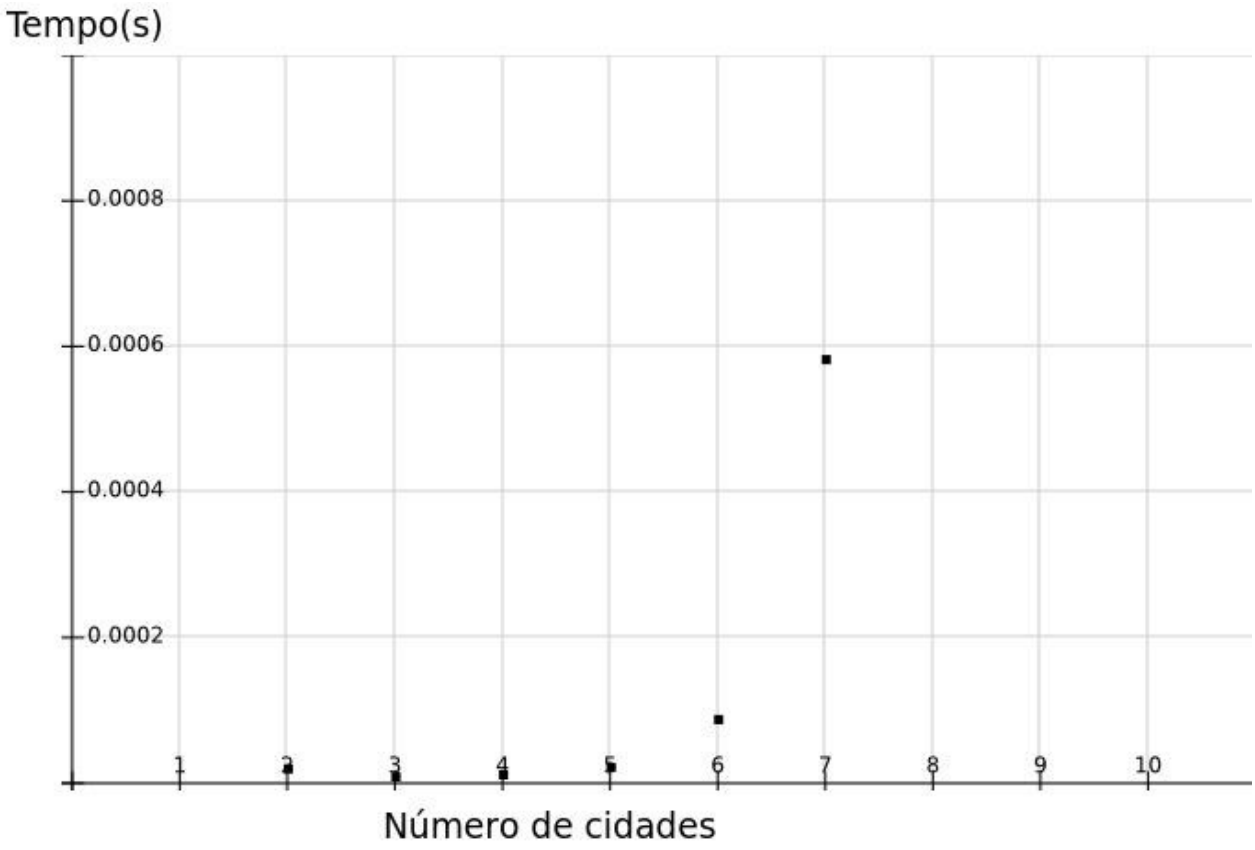

Gráfico 2: Número de cidades x Tempo de execução com zoom, mostrando pontos de 2 até o 7

Podemos ver que o tempo de execução "explode" rapidamente, e após o quinto vértice, o tempo de execução para uma quantidade de cidades aumenta muito em comparação ao tempo levado pelo número anterior de cidades, e a própria visualização do gráfico fica comprometida. Para facilitar a visualização, foram feitos dois gráficos: um com os dados relativos aos testes com quantidade de cidades variando de 2 até 7, e outro incluindo os casos com 8 ou mais cidades. Podemos observar que esse aumento grande no tempo de execução de uma entrada para outra se deve ao fato de que a complexidade da função que encontra o menor caminho permitido tem complexidade exponencial (mais especificamente fatorial): **O(n!)**. Os dados corroboram com a análise experimental feita.

5. Conclusão

Ao fim do trabalho, pudemos compreender a dificuldade em resolver problemas pertencentes à classe NP-Completo. No programa desenvolvido, até mesmo casos de teste com menos de vinte cidades demoraram um tempo considerável para serem executados. O caso de teste com treze cidades levou aproximadamente 18 minutos para executar. Seguindo o padrão de tempo fatorial para execução de uma entrada, obtido tanto na análise de complexidade quanto na análise experimental, temos que um caso de teste com 14 cidades demoraria aproximadamente quatro horas para executar, e um caso de teste com 15 cidades demoraria mais de dois dias para completar sua execução! Para permitir a execução do programa em determinados casos onde a quantidade de cidades se aproxima mais do limite superior permitido (22 cidades), nos utilizamos de podas, que consistem em abortar buscas de caminhos que obviamente não serão ótimos. A

poda mais simples implementada consiste em abortar caminhos em que uma cidade B aparece antes de outra cidade A, tão antes quanto B apareça, se soubermos que A contém produtos que devem ser entregues em B. Outra poda consiste em abortar caminhos que já ultrapassaram o tamanho do menor caminho já encontrado. Assim, permitimos que uma maior variedade de casos seja resolvida em tempo hábil.

Os resultados obtidos na análise experimental concordam com os obtidos na análise de complexidade.