Documentação TP2 **Ícaro Harry**
AEDS3 Matrícula: 2014033050

1 Introdução

O trabalho proposto no TP2 foi a resolução do Problema do Caixeiro Viajante, utilizando uma abordagem de "podas" (estratégias de *branch and bound*). Para isso, são dadas restrições de precedência para a visitação de cidades, onde se existe uma restrição entre A e B, A deve ser visitada **antes** do que B. O problema, em sua versão de decisão, consiste em um clássico NP Completo. Ou seja, não existe solução em tempo polinomial conhecida. Isso faz com que resolver instâncias relativamente pequenas do problema demande um poder de processamento inexistente na atualidade.

A ideia das estratégias de *branch and bound* consiste em limitar (podar) a árvore de possíveis soluções do problema, eliminando a necessidade de verificar certos caminhos. Dessa maneira, a solução que desenvolvi para o problema, executando podas, é capaz de executar determinadas instâncias do PCV em um tempo razoável.

2 Modelagem do Problema

O problema proposto é um clássico PCV (Problema do Caixeiro Viajante), onde todas as cidades possuem ligações entre si. A única diferença no enunciado é a existência de *restrições de precedência*, as quais determinam que algumas cidades devem ser visitadas antes do que outras.

2.1 Grafo de cidades

Utilizei o método de **matriz de adjacências** para implementar os grafos.

As cidades são representadas por meio de um **grafo não-direcionado completo**, onde as arestas representam as distâncias entre as cidades. Esse grafo não é direcionado pois, pela descrição do problema, não existe diferença entre os caminhos de ida e volta entre duas cidades.

2.2 Grafo de restrições

As restrições são representadas por um **grafo direcionado sem peso**, onde uma aresta entre o vértice **A** e o vértice **B**, representa uma restrição de que **A** deve ser vistada antes do que **B**. Porém, existe a necessidade de que esse grafo seja direcionada, já que existe uma diferença entre uma restrição de A para B e de B para A.

2.3 Caminhamento pelas cidades

O Problema do Caixeiro Viajante consiste em percorrer **todos** os (n-1)! possíveis caminhos para verificar qual o menor deles. Dessa maneira, para percorrer todos esses caminhos, necessita-se de um algoritmo que permute todas as possíveis ordens de visita aos vértices. Para fazer isso, utilizei o algoritmo de *DFS (Depth First Search, ou Busca em Profundidade)*, modificado.

O algoritmo de busca em profundidade, em sua versão recursiva, percorre um caminho até o final e na volta, percorre os vértices adjacentes, recursivamente. Além disso, por meio de um vetor *booleano*, o algoritmo verifica quais vértices já foram visitados, para não percorrê-los novamente. Todavia, para que todos os possíveis caminhos sejam visitados, é necessário que os vértices que foram marcados como visitados, sejam desmarcados, ao final da recursão, para que possam ser visitados novamente, dessa vez em outra ordem.

Algorithm 1 ModifiedDFS

```
function DFS(G, v, n, visited)
    visited[v] = true;
    for i = 0; i minor than n; i++ do
        if !visited[i] then
            DFS(G, i, n);
            visited[i] = false;
        end if
    end for
end function
```

2.4 Métodos de Branch and Bound (podas)

Como dito anteriormente, para localizar o menor caminho, deve-se percorrer todas as possíveis combinações de vértices, o que torna o algoritmo fatorial (O(n!)). Dessa maneira, para o número máximo de cidades do problema (22), é necessário verificar 21! caminhos, o que demoraria anos para ser executado em um PC comum. Todavia, é possível reduzir a árvore de possíveis soluções, utilizando técnicas de *podas*, onde, dada determinada configuração, é possível concluir que ela não levará ao resultado ótimo, e assim, é possível eliminá-la antes de checá-la até o final, evitando-se visitar diversos vértices. Sendo assim, descreverei todas as podas que realizei em minha solução.

É importante destacar que, além de podas que são "intrínsecas" ao PCV, as restrições de precedência propostas no enunciado, permitem podar ainda mais a árvore de possíveis soluções. Já que, ao encontrar uma aresta que desrespeita uma restrição, não é necessário continuar visitando os vértices após essa aresta.

2.4.1 Poda do caminho atual - *(Poda 1)*

A poda mais simples realizada é a que verifica se o menor caminho já visitado é menor ou igual ao caminho que se está verificando atualmente. Se isso ocorrer, o caminho atual

não é promissor, ou seja, é impossível que ele seja o menor de todos.

É simples realizar essa poda. Basta armazenar em uma variável o menor caminho já encontrado até agora e compará-lo ao caminho percorrido, caso a aresta até o próximo vértice a ser visitado, somada ao caminho atual, iguale ou ultrapasse o menor caminho atual, não se chama a recursão para aquele vértice.

Algorithm 2 ModifiedDFS with the first branch and bound

```
function DFS(G, v, n, visited, current, minimum)
    visited[v] = true;
    for i = 0; i minor than n; i++ do
        if !visited[i] then
            if (current + G[v][i] minor than minimum) then
                DFS(G, i, n);
                visited[i] = false;
            end if
        end if
    end for
end function
```

2.4.2 Poda do caminho da volta - *(Poda 2)*

Outra poda que pode ser realizada é a poda de eliminar o caminho da volta. Como o grafo não é direcionado, temos que a aresta de A para B é igual a aresta de B para A. Dessa maneira, ao permutar todos os possíveis caminhos, o caminho da ida e o caminho da volta vão ser verificados, mas na prática, eles representam o mesmo caminho. Por exemplo, dada as cidades A, B, C, D e E o caminho ABCDEA é igual ao caminho AEDCBA. Entretando, quando existe uma restrição de B para C, todos os caminhos em que C aparece antes de B, podem ser descartados, o que reduz pela metada a árvore de caminhos. Como mostrado nos exemplos:

Caminhos onde B aparece depois de C: CBDE, CBED, CDBE, CDEB, CEBD, CEDB, DCBE, DCEB, DECB, ECBD, ECDB, EDCB.

Caminhos onde B aparece antes de C: BCDE, BCED, BDCE, BDEC, BEDC, BECD, DBEC, DBCE, DEBC, EBCD, EBDC, EDBC.

Portanto, se não existe nenhuma restrição de precedência já informada na entrada, basta criar uma restrição de B para C.

2.4.3 Podas das restrições

As restrições permitem realizar várias podas na árvore de caminhos. Para verificar se o caminho ainda é promissor, fiz uma função que retorna true se o caminho é restrito (não promissor) e false se é promissor. Dessa maneira, alterei o *if* que checa a primeira restrição, para a seguinte maneira:

Algorithm 3 ModifiedDFS with the first branch and bound and restrictions

```
if (current + G[v][i] minor than minimum AND !isRestricted(i, v, restrictions, n))
then
    DFS(G, i, n);
    visited[i] = false;
end if
```

A primeira poda relacionada às restrições *(Poda 3)* é a mais simples a ser feita. Ela consiste em não ir do vértice **A** para o vértice **B** se existe uma restrição de **B** para **A**. Pois, caso exista essa restrição, é necessário que o vértice **B** seja visitado antes do que A, então todos os caminhos em que **A** apareça antes do que **B**, podem ser descartados.

Algorithm 4 Restrictions verification

```
function isRestricted(a, b, restrictions)
    if (restrictions[a][b]) then
        return true
    else
        return false;
    end if
end function
```

A segunda poda relacionada às restrições *(Poda 4)* consiste em descartar todos os caminhos que desrespeitam uma restrição de precedência, porém não só excluindo os caminhos em que a precedência é adjacente, como na *Poda 3*. Por exemplo, dadas as cidades W, X, Y e Z, se existe uma restrição de Y para X. A poda 3 irá descartar os caminhos WXYZW e WZXYW, porém não irá descartar o caminho WXZYW, o qual a poda 4 irá descartar.

Essa poda ocorre apenas quando a poda 3 não ocorre, pois ela é capaz de detectar também as restrições que a poda 3 detecta, todavia em tempo linear, enquanto a poda 3 é constante. E se a poda 3 ocorre, não é necessário verificar outras quebras de restrição, pois o caminho já pode ser descartado.

Ela consiste em percorrer todos os vértices adjacentes ao vértice que será visitado. Se existir uma restrição entre eles e o vértice adjacente não foi visitado, então ocorreu uma quebra de restrição. Para fazer isso, modifiquei a função *isRestricted*.

Algorithm 5 Restrictions verification incremented

```
function isRestricted(a, b, restrictions, n, discovered)
    if (restrictions[a][b]) then
        return true
    else
        for i = 0; i minor than n; i++ do
            if restrictions[i][a] AND !discovered[i] then
                return true
            end if
        end for
    end if
    return false;
end function
```

2.4.4 Podas do Deadlock

Dadas as restrições fornecidas no enunciado do problema, podem ocorrer casos em que é impossível respeitá-las, por exemplo. A necessita ser visitada antes do que B, B necessita ser visitada antes do que C e C necessita ser visitada antes do que A. Por transitividade, temos que: A deve ser visitada antes do que C e C deve ser visitada antes do que A, o que é impossível e representa um *Deadlock*.

Dessa maneira, a primeira poda relacionada aos Deadlocks *(Poda 5)* é bem simples: dado uma restrição de A para B, verificar se já existe uma restrição de B para A. Caso isso ocorra, imprimir Deadlock.

A segunda poda relacionada aos Deadlocks *(Poda 6)*, consiste em verificar se existe uma situação como a citada no exemplo, onde por transitividade, é possível concluir que existe um Deadlock. Pode-se verificar isso checando se existe um ciclo no grafo de restrições, pois o ciclo representa justamente um ciclo de dependências, que torna impossível a existência de uma solução para a entrada.

O algoritmo para verificar se existe um ciclo em um grafo direcionado pode ser obtido também por uma modificação da busca em profundidade. Basta percorrer o grafo começando de V e toda vez que o vértice W for visitado, marcá-lo na "pilha de recursão". E caso um vértice que será visitado, estiver na pilha de recursão, isso representa um ciclo. Como demonstrado abaixo:

Algorithm 6 ModifiedDFS to verify if there's a cycle

```
function verifyCycle(G, v, n, visited, onStack, hasCycle)
    visited[v] = true;
    onStack[v] = true;
    for i = 0; i minor than n; i++ do
        if G[v][i] then
            if !visited[i] then
                verifyCycle(G, v, n, visited, onStack, hasCycle)
            else if onStack[i] then
                hasCycle = true;
            end if
        end if
    end for
    onStack[v] = true;
end function
```

2.4.5 Diagrama de podas

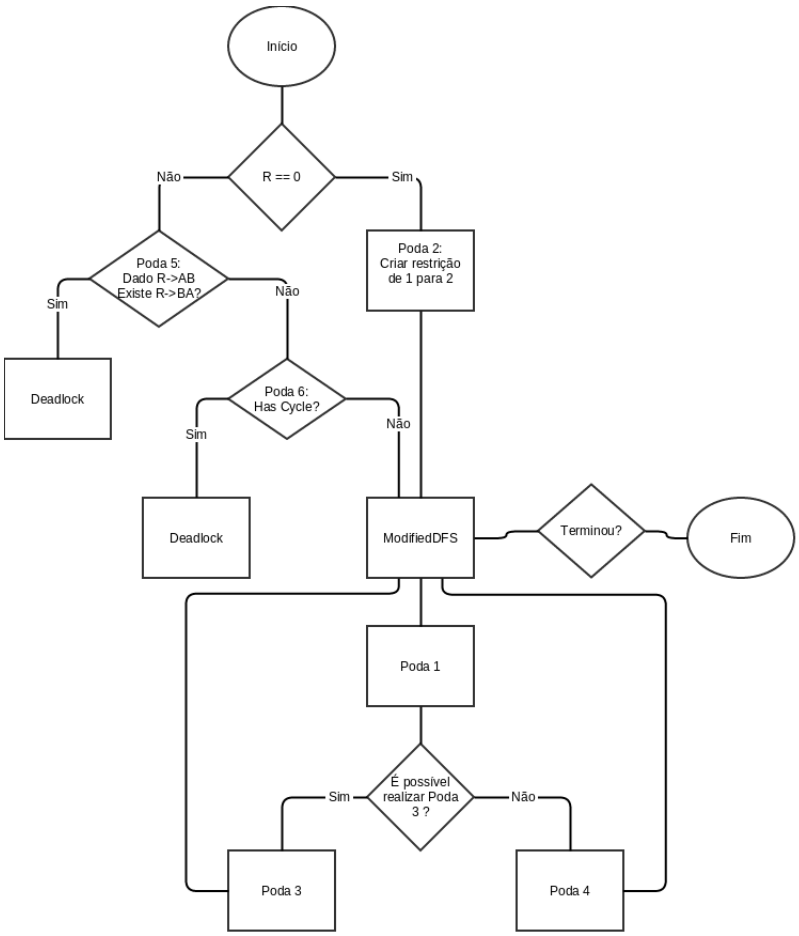

Figure 1: Diagrama de podas

3 Análise Teórica do Custo Assintótico de Tempo

O Problema do Caixeiro Viajante, como já citado acima, possui complexidade fatorial. Para demonstrar isso, será provado abaixo que a sua versão de decisão (versão sim-não) faz parte dos chamados problemas **NP-completo**.

3.1 Prova NP-completo para o Problema do Caixeiro Viajante

O Problema do Caixeiro Viajante, em sua versão de decisão, pode ser descrito como: dado um grafo completo **G**(v, e) e uma constante **k**. É possível sair do primeiro vértice, percorrer todos os outros, sem repetição, e voltar ao primeiro com custo menor ou igual a **k**?

Em primeiro lugar, é necessário demonstrar que mesmo com todas as restrições, o problema ainda pode cair em um caso que necessite de de percorrer (n - 1)! caminhos, logo ainda é exponencial. Pelo Diagrama de Podas demonstrado acima, pode-se perceber que é possível que não haja restrições, logo será criada uma restrição que irá reduzir pela metade o tamanho da árvore de caminhos. Dessa maneira existem (n-1)!/2 caminhos a serem percorridos. Além disso, a solução pode ser o último caminho verificado, o que impediria as podas 1, 3 e 4, sendo esse o pior caso. Ainda seria necessário verificar (n - 1)!/2 caminhos.

3.2 Prova NP

Seja $C = \{V1, V2, ..., Vj, V1\}$, uma solução para o PCV. O seguinte algoritmo verifica se C é uma solução para PCV em G.

O algoritmo acima possui complexidade linear, pois percorre os vértices do grafo 2 vezes, logo possui complexidade O(v).

3.3 Prova NP-completo

Para provar que PCV pertence a NP Completo, utilizarei o problema do Caminho Hamiltoniano. Que é descrito a seguir:

Dado o grafo G(v, e) e um vértice V0, é possível partir de V0, percorrer todos os vértices, sem repetição, e retornar a V0?

3.3.1 Transformação

Seja G(V, E) uma instância do CH, construiremos uma instância G'(V, E') do PCV, a partir dela, da seguinte maneira: Se existe uma aresta de Va para Vb em G, então criar uma aresta Va para Vb em G' com peso 0. Caso contrário, criar uma aresta de Va para Vb com peso 1.

Algorithm 7 Verify TSP solution

```
function verifySolution(G, C, k, n)
    ▷ Se C possuir menos elementos do que o número de vértices de G + 1 (A aresta que volta a V1) então é uma solução inválida.
    if C.numberOfElements() != G.numberOfVertex() + 1 then
        return false;
    end if
    ▷ Verifica se há repetições de vértices e se todos os vértices fazem parte do grafo
    Seja aux um vetor booleano
    for i = 0; i minor than n; i++ do
        if !aux[C[i]] then
            aux[C[i]] = true;
        else
            return false;
        end if
    end for
                                                    ▷ Verifica se o caminho é menor do que K
    for i = 0; i minor than n - 1; i++ do
        path += G[i][i + 1];
    end for
    path += G[n - 1][0];
    if path minor or equal k then
        return true;
    else
        return false;
    end if
end function
```

3.3.2 Prova

Seja h uma viagem em G. Pela construção de G', ela possui custo 0 no grafo G'. Logo é uma viagem em G' com custo 0.

Por outro lado, seja m uma viagem em G' com custo máximo 0. Tendo em vista a construção do grafo G', a única maneira de obter custo 0, é percorrer o Ciclo Hamiltoniano.

Dessa maneira, G tem um Ciclo Hamiltoniano **se e somente se** G' tem uma viagem de custo máximo 0 e G é solução para CH **se e somente se** G' é solução para G'.

4 Análise Teórica do Custo Assintótico de Espaço

O programa necessita de pouco espaço em memória para ser executado. No início são alocadas duas matrizes de 23x23 para as cidades e as restrições.

4.0.3 ModifiedDfs

A principal função do programa, utiliza dois vetores de V posições e, como é recursiva, mantém uma pilha de recusão do tamanho de V. Sendo assim, possui custo linear O(V).

5 Análise Experimental do Custo Assintótico

Para verificar o custo assintótico, realizei testes utilizando entradas que caiam no pior caso, descrito na seção **3.1**. Para medi-los utilizei a função time do ubuntu.

5.0.4 Análise Experimental

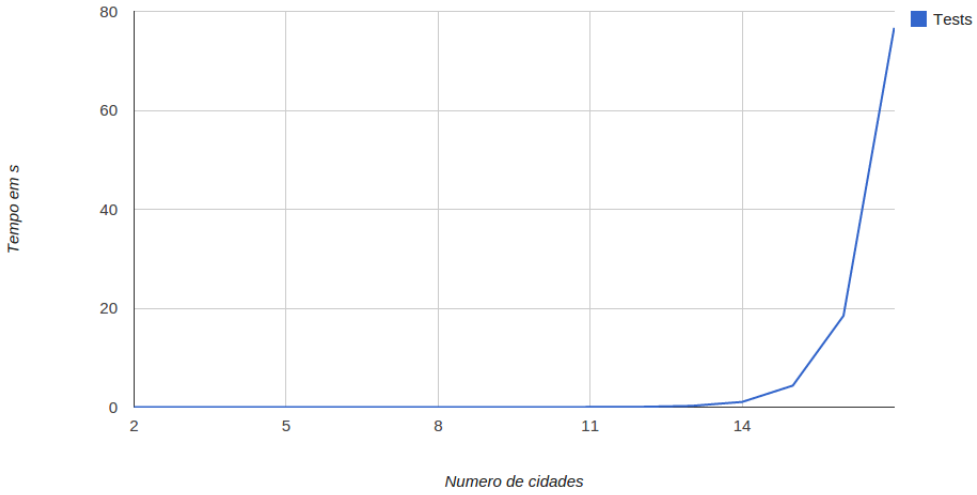

Figure 2: Testes realizados

6 Conclusão

A execução de problemas NP-completo pode ser muito custosa devido a sua natureza exponencial. Todavia, é possível, de acordo com as características das variações do problema, criar melhorias que permitem executar diversos casos em curto tempo.

As técnicas de Branch and Bound realizadas nesse trabalho permitiram a execução de diversas instâncias do PCV que com um simples algoritmo de força bruta são impossíveis de serem realizadas.