Alunos: Kevyn Menezes Carvalho, Daniel da Silva Teodoro, Abner da Silva Tavares

Neste trabalho apresento como encontrar uma solução para o problema do clique através de uma redução para o problema do vertex cover.

Como demonstrado na *Figura 1*, os vértices são números que variam de 0 até n-1.

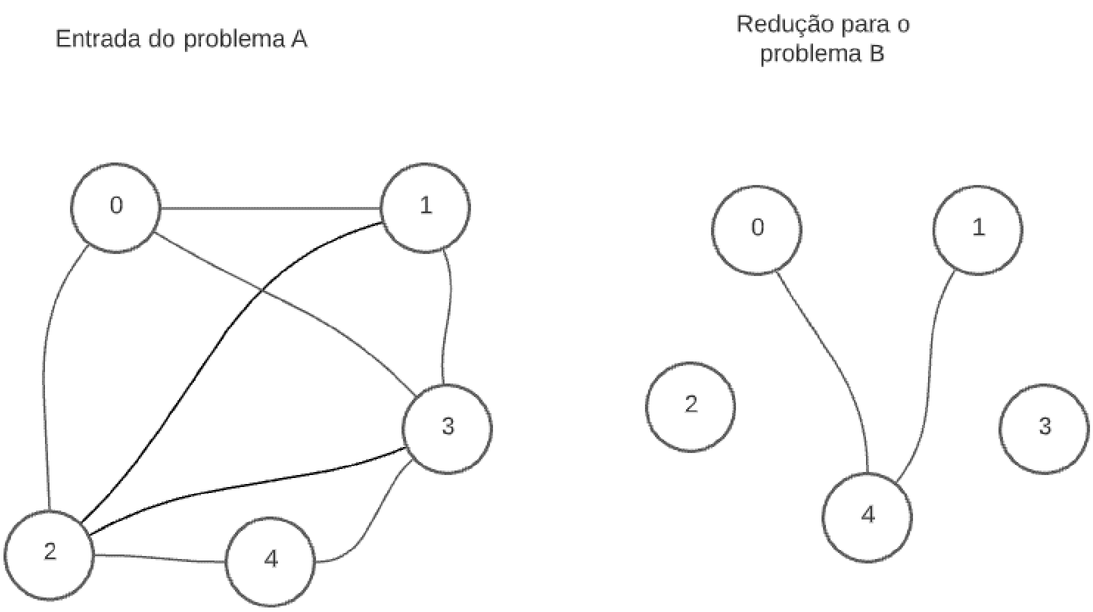

Figura 1. Representação gráfica do grafo.

Representação de um grafo em C:

Foi utilizado a linguagem C para escrever a função que encontra uma solução para o clique através de redução. Foi usado uma lista estática para representar os vértices, e para cada vértice foi usado uma lista ligada para representar as arestas (vértices vizinhos), a *Figura 2* é uma representação de como o grafo da *Figura 1* fica em memória no código em C.

```
|0| -> [1] -> [2] -> [3] -> [NULL]
|1| -> [0] -> [2] -> [3] -> [NULL]
|2| -> [0] -> [1] -> [3] -> [4] -> [NULL]
|3| -> [0] -> [1] -> [2] -> [4] -> [NULL]
|4| -> [2] -> [3] -> [NULL]
```

Figura 2. Representação de como um grafo fica armazenado na memória em C

Entrada para o grafo do problema A.

Para a entrada do problema A, foi lido um arquivo *graph.txt* que contem todas as arestas do grafo G, cada linha é uma aresta do grafo G. A *Figura 3* é uma representação em txt dos vértices 0 e 1 do grafo da *Figura 1.*

```
0 1
0 2
0 3
1 0
1 3
1 2
```

Figura 3. Representação em txt de 2 vértices do grafo da Figura1.

Função que acha a solução para o problema do clique através de redução.

```
Solution *clique_solution(Graph *G)
```

Esta função encontra uma solução para o problema do clique reduzindo o problema do clique em um problema do vertex cover através da função *graph_complement*. Em seguida, acha uma solução para o problema do vertex cover através da função *vertex_cover_solution.* E, por fim, converte a solução do vertex cover para uma solução do clique através da função *convert_vertex_solution_to_clique*.

Função de redução de A para B:

```
Graph *entrada_vertex = graph_complement(G);
```

É uma função que dado um grafo G retorna um grafo S, cujo S é o complemento de G. Então S será uma entrada para o problema B.

Essa função percorre todos os vértices V do grafo G e para cada V_i verifica se V_J não é vizinho de V_i, caso não seja, adiciona-se no grafo S, no vértice V_i um vizinho V_j.

```
void add_edge(Graph* G, int V, int E)
```

O custo para inserir um vizinho em um determinado vértice do grafo é de no pior caso $n-2$, pois em um vértice V podem existir até $n-1$ vizinhos. Então, se o vértice V tem $n-2$ vizinhos conectados, é preciso percorrer todos os vizinhos anteriores, pois neste algoritmo se faz uma inserção ordenada em uma lista ligada. Isto poderia ser reduzido a $O(1)$ caso não fosse importante guardar os vizinhos de forma ordenada. Então, pode-se dizer que a função *add_edge* tem um custo de $O(n)$.

Em um grafo totalmente desconexo, o grafo complemento S será um grafo completo, então a função *add_edge* com custo O(n) será executada $n(n-1)$ vezes na função *graph_complement* para adicionar as arestas em S, assim pode-se dizer que esta função tem uma complexidade de $O(n^3)$.

Função que encontra uma solução ótima para o problema B:

```
Solution *solution_vertex_cover(Graph *G)
```

Para encontrar uma solução para o problema B, foi usado um algoritmo que testa todas as combinatórias de vértices usando a função *verify_solution* para verificar se uma combinatória é uma solução válida e decrescendo o número do tamanho da solução a cada solução encontrada. O algoritmo para quando não encontra uma solução de tamanho k, ou quando k = 0. Então infere-se que a solução ótima é de tamanho k+1.

```
int verify_solution(Graph *G, Solution *s)
```

Esta função verifica se uma combinatória é válida para o problema do vertex cover em cima de um grafo G, retornando 1 caso seja uma solução válida e 0 caso não seja.

Essa função funciona da seguinte forma:

Uma suposta solução é uma estrutura contendo o conjunto de vértices, junto com o tamanho da solução. Neste conjunto de vértices, os vértices que estão marcados são representados com o numero 1, e os que não estão marcados são representados com o número 0.

Dado uma solução S em cima de um grafo G, percorre-se por todos os vértices V no grafo G que não estão marcados na solução S. Então percorre-se por todos os vizinhos de V e se algum vizinho de V não estiver marcado na solução S, S não é uma solução para G. Caso para todo V não marcado, todos os vizinhos de V estejam marcados, então S é uma solução para o grafo G. Isso é verdade, pois se existe algum vizinho "E" não marcado em um vértice V que também não esteja marcado, a ligação entre V e "E" não está sendo coberta, e se existir pelo menos uma aresta não esteja coberta no grafo G, S não será uma solução para G.

Então para verificar uma solução com a função *verify_solution* no grafo G, precisamos percorrer, no pior caso, por todas as arestas do grafo G. Se G for um grafo completo, que também é um dígrafo, este terá $n(n-1)$ arestas, então pode-se inferir que a complexidade de *verify_solution* é de $O(n^2)$.

O número total de combinatórias a ser testada no pior caso pode ser expressado pela seguinte equação: $\sum_{x=1}^{n-1} \frac{n!}{x!(n-x)!}$ Onde x é o tamanho da solução que itera de $n-1$ até 1. O algoritmo testa todas as combinatórias possíveis com $x = n-1$, se é encontrado alguma solução, o algoritmo passa a verificar as combinatórias com $x = n-2$, isso acontece até ele não achar uma solução com tamanho k, então infere-se que a solução ótima é de tamanho k+1, assim retornando a ultima solução encontrada. Como o algoritmo testa se uma solução é válida com a função *verify_solution* pode-se dizer que, no pior caso, o algoritmo que acha a solução para o vertex cover vai ter uma complexidade de $n^2.\left(\sum_{x=1}^{n-1} \frac{n!}{x!(n-x)!}\right)$, então pode-se dizer que a complexidade do algoritmo que encontra a solução para o problema do vertex cover é de $O(n!)$.

Função que converte a solução do problema B para uma solução do problema A.

```
void convert_vertex_solution_to_clique(Solution *s)
```

Dada uma solução ótima para o problema B, tudo que é preciso fazer é pegar o complemento dessa solução, ou seja, todos os vértices que estão na solução para B não estarão na solução para A. E todos os vértices que não estão na solução para B, estarão na solução para A. Uma é o complemento da outra. O tamanho da solução para A é a diferença entre o número total de vértices e o tamanho da solução para B. Dito isto, tudo que é preciso fazer é pegar a solução para B e para cada vértice V nesta solução, nega-se V_i, assim qualquer vértice que esteja na solução para B, não estará na solução para A, e qualquer vértice que não esteja na solução para B, agora estará na solução para A. Como esse algoritmo só percorre o vetor de vértices uma única vez, então a complexidade dessa função é $O(n)$.

Então temos uma complexidade de $O(n^3)$ para converter uma entrada do problema A para uma entrada do problema B. Temos também uma complexidade de $O(n!)$ para achar uma solução para B. E, por fim, uma complexidade de $O(n)$ para converter a solução de B para A. Sendo assim, temos que para encontrar a solução do problema A através de redução a complexidade é de $O(n!)$.