**(5)**

**a)** Problemas pseudopolinomiais são problemas que podem ser resolvidos por um algoritmo determinista em tempo polinomial quando os valores de entrada tem tamanho polinomial, mas apresentam complexidade exponencial de tempo quando a entrada tem tamanho exponencial (por exemplo o número de bits necessários para representar a entrada).

Um exemplo clássico de problema pseudopolinomial é o problema da soma dos subconjuntos (*subset sum).* Neste problema tem-se um conjunto de números e um valor objetivo e deseja-se saber se é possível descrever o valor objetivo como uma soma de algum subconjunto desse conjunto de números. Quando os números dados são pequenos (cabem em uma variável inteiro, por exemplo), pode-se resolver esse problema em tempo polinomial O(tn) usando um algoritmo de programação dinâmica, onde t é o valor objetivo e n é a quantidade de números do conjunto original. Por outro lado, se o objetivo dado e/ou os números do conjunto original forem inteiros grandes (não cabem em uma variável de inteiro) a complexidade se torna exponencial O(2bn), onde b é o número de bits do número t.

**b)** Para provar que um problema pertence a classe de problemas NP é necessário provar que existe um algoritmo não determinista que o resolve em tempo polinomial ou então um usar a noção de certificado, ou seja, dada uma instância do problema e uma solução, mostrar que existe um algoritmo determinista que verifica se a solução é válida ou não em tempo polinomial. Logo, apresenta-se abaixo um algoritmo certificado que recebe um grafo (em forma de matriz de adjacência), um inteiro V indicando o número de vértices, um vetor V’ indicando a solução a ser validada e um inteiro k indicando o custo máximo aceitável para esta instância do caixeiro viajante:

|  |
| --- |
| **func** certificado(Grafo G, Inteiro V, Vetor V’, Inteiro k):  Vetor viz[v]  **Para cada** v ∈ V: viz[v] = não-marcado  Inteiro custo = 0  Inteiro n = | V’ | // |V’| é o tamanho da solução  **Se** V’[0] ≠ V’[n-1]: **retorne** falso // não é um ciclo    **Para** i=0 até n-2:  // Se um vértice já foi visitado uma vez  **Se** viz[ V’[i] ] == marcado: **retorne** falso  **Senão**: viz[ V’[i] ] = marcado  custo = custo + G[ V’[i] ][ V’[i+1] ].custo    **Se** custo > k: **retorne** falso // custo maior do que o aceitável  **Para** i=0 até n-1: // verifica se todos os vértices foram vizitados  Se viz[ i ] == não-marcado: **retorne** falso  **retorne** verdadeiro |

Analisando o algoritmo acima é possível perceber que este algoritmo certificado tem complexidade O(V + |V’|) onde V é a quantidade de vértices do grafo e |V’| é o tamanho da solução a ser testada. Ou seja, trata-se de um algoritmo com complexidade linear (polinomial) e portanto pode-se afirmar que o problema do caixeiro viajante está em NP.

**c)** Para provar que o problema de cobertura de vértices pertence ao conjunto de problemas NP-Completo é necessário fazer duas provas: (1) mostrar que o problema pertence a NP; (2) mostrar que é possível reduzir um problema NP-Completo conhecido ao problema de cobertura de vértices.

Para a primeira prova, mostrar que o problema de cobertura de vértices pertence a NP, usamos novamente a ideia de certificado – um algoritmo determinista que recebe uma instancia do problema e uma solução, e verifica se a solução é válida ou não em tempo polinomial. Logo, apresenta-se abaixo um algoritmo certificado que recebe um grafo (em forma de lista de adjacências), um vetor V’ indicando a solução de vértices pertencente a cobertura a ser validada e um inteiro k indicando o número máximo aceitável de vértices que compõem a resposta:

|  |
| --- |
| **func** certificado(Grafo G, Vetor V’, inteiro k):  **Se** |V’| > k: **retorne** falso // a solução é maior do que o desejado  **Para** cada (u,v) Adjacências:  // Verificar se a aresta é coberta por uma de suas extremidades  **Se** u V’ and v V’: **retorne** falso  **retorne** verdadeiro |

Analisando o algoritmo acima e considerando que a operação de busca que verifica se u e v não pertencem ao vetor de solução pode ser implementada com complexidade linear (varrer o vetor V’ procurando se u ou v não estão lá dentro), é possível perceber que este algoritmo certificado tem complexidade O(E|V’|) onde E é a quantidade de arestas do grafo e |V’| é o tamanho da solução a ser testada. Ou seja, trata-se de um algoritmo com complexidade polinomial e portanto pode-se afirmar que o problema está em NP.

Para a segunda prova, precisamos reduzir um problema NP-Completo conhecido no problema de cobertura de vértices em tempo polinomial. O problema NP-Completo escolhido é o problema do CLIQUE (encontrar o maior subgrafo completo em um grafo). Apresenta-se abaixo um algoritmo de redução que recebe um grafo (em forma de matriz de adjacência), o número de vértices deste grafo, e o transforma em um grafo equivalente para o problema da cobertura de vértices (grafo inverso):

|  |
| --- |
| **func** redutor( Grafo G, Inteiro V ):  Grafo novo(V) // cria um grafo/matriz V\*V  **Para** i=0 até V:  **Para** j=i+1 até V:  // Se existe uma aresta de i para j -> agora não existe  **Se** G[i][j] == 1:  novo[i][j] = novo[j][i] = 0;  // Se não existe uma aresta de i para j -> agora existe  **Senão**:  novo[i][j] = novo[j][i] = 1;  **retorne** novo |

É fácil perceber que o algoritmo acima executa com complexidade O(V²) onde V é o número de vértices do grafo do problema do CLIQUE. Note que o grafo resultante nada mais é do que o grafo inverso ao grafo original (ou seja, onde tinha aresta não tem mais, e onde não havia aresta, agora há).

**d)** definir problema conjunto independente de vértices máximo, formule a versão de decisão do problema e prova que pertence a NP-Completo

[não conhecia este problema... não é tão usual como os demais]

**e)** Tem um exemplo de como fazer a redução aqui:

<https://npcompletoblog.wordpress.com/3-coloring/>

**f)** A verificação de um grafo bipartido se relaciona com a variante 2-Coloração onde temos apenas duas cores para colorir o grafo. Na prática estes problemas têm objetivos semelhantes com mínimas diferenças, sendo que ao resolver um deles, temos a solução para o outro. Este objetivo pode ser descrito como manter dois subgrupos de vértices de um grafo sem intra-ligações (ligações entre nós do mesmo subgrupo) e possíveis inter-ligações (ligações entre nós de um subgrupo diferente).

Como o problema do grafo bipartido não é NP-Completo, o problema 2-Coloração também não é NP-Completo (semelhante ao k-SAT, onde 2-STA não é NP-Completo, mas 3-SAT é). Para estes problemas, uma solução clássica se utiliza do conceito de fluxo em grafos – Edmonds-Karp. Este algoritmo tem complexidade de tempo O(VE²). Atualmente já existem algoritmos mais eficientes, mas este algoritmo já mostra que o problema pode ser resolvido em tempo polinomial.

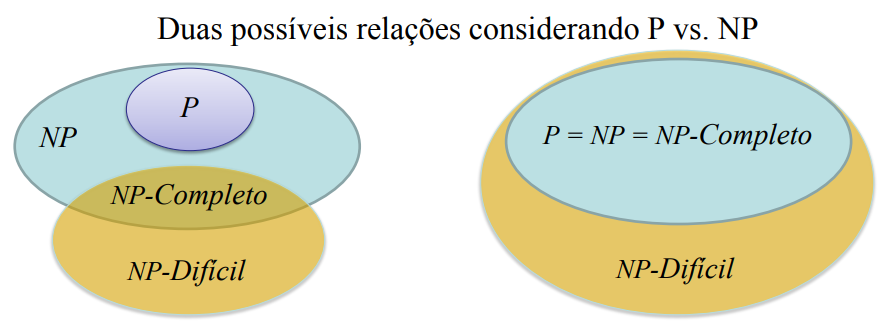
**g)** Algoritmos de aproximação, heurísticas e metaheurísticas são algoritmos que tentam encontrar soluções ótimas em um tempo aceitável, mas não garantem que as vão encontrar, podendo gerar soluções ruins, medianas, boas e porventura ótimas. Geralmente estas abordagens são indicadas quando um problema é muito complexo para ser resolvido por um algoritmo que busca a solução ótima que demora muito tempo. Cada uma dessas abordagens tem algumas características específicas.

As heurísticas realizam procedimentos que geralmente são específicos para um dado tipo de problema e não fornecem nenhuma garantia em relação ao resultado final. Geralmente possuem alguma inspiração lógica, ou da natureza (ex: *simulated annealing*) e em alguns casos podem ser elaborados como algoritmos gulosos, os quais não encontram a solução ótima, mas podem encontrar soluções boas em um tempo aceitável.

Os algoritmos de aproximação são parecidos com as heurísticas, sendo geralmente específicos para um dado tipo de problema, mas a grande diferença desses algoritmos é que eles oferecem uma garantia sobre o resultado final encontrado na forma de inequação. Por exemplo, existe um algoritmo de aproximação para o problema do caixeiro viajante que garante que a solução encontrada é no máximo duas vezes pior do que a melhor solução possível. Geralmente esse tipo de algoritmo se baseia em alguma propriedade matemática, lógica ou do problema.

Por fim as metaheurísticas são algoritmos caixa preta que podem ser facilmente adaptados para diferentes tipos de problema. Existem muitos desses algoritmos, principalmente inspirados na natureza: colônia de formigas, colônia de abelhas, otimização por enxame de partículas, algoritmo genético, programação genética, evolução diferencial etc.

**h)** Para as questões abaixo, considerar as seguintes possibilidades de conjuntos:



Lembre-se que:

**classe P** = problemas que podem ser resolvidos por algoritmos deterministas (algoritmos normais / que estamos acostumados) em tempo polinomial

**classe NP** = problemas que podem ser resolvidos por algoritmos não-deterministas (algoritmos que sempre escolhem o caminho correto para chegar a uma solução) em tempo polinomial

**classe NP-Completo** = são os problemas mais difíceis em NP. Se alguém provar que existe um algoritmo determinista que resolve um problema NP-Completo em tempo polinomial, então será possível afirmar que P = NP.

**Classe NP-Difícil (*NP-Hard*)** = são os problemas mais difíceis que existem. Eles podem ser tratáveis ou intratáveis (nunca se encontra todas as soluções que se deseja). Os problemas tratáveis podem ser resolvidos por algoritmos não deterministas em tempo polinomial (são os problemas NP-Completo) ou apenas em tempo exponencial (problemas que estão fora do conjunto NP).

i) Falso. Na verdade depende do tipo de problema NP em questão. Se este problema NP, estiver em NP-Completo (ou seja, não deixa de ser um problema NP), ao resolvê-lo com um algoritmo determinista em tempo polinomial estaria se provando que P = NP. Por outro lado, se o problema fosse resolvido por um algoritmo não-determinista em tempo polinomial, estaria se provando apenas que o problema realmente pertence a NP. Já se o problema NP em questão, não estiver em NP-Completo e for resolvido em tempo polinomial por um algoritmo determinista, só prova que o problema está no conjunto de problemas P (e portanto, também é NP).

ii) Verdade. Mas depende do tipo de algoritmo utilizado. Para garantir que P=NP, o algoritmo utilizado deve ser determinista. Se for um algoritmo não determinista, só mostra que o problema NP-Difícil está no conjunto NP-Completo.

iii) Falso. Existem problemas em NP-Difícil que são intratáveis (ou seja, não são resolvíveis) e problemas que são resolvidos em tempo exponencial ou maior.

iv) Falso. Não se pode afirmar isso, pois ninguém conseguiu provar ainda que não se pode existir um algoritmo determinista que resolva um problema NP-Completo. Mas a maior parte da comunidade acredita que esse é o caso e por isso P != NP. Essa crença se dá pela grande quantidade de problemas NP-Completos existentes e pela incapacidade, até o momento, de se desenvolver um algoritmo determinista que resolva apenas um desses problemas em tempo polinomial.

v) Falso. Para provar que P = NP, a redução a ser feita deveria ser contrária: P2 ≤p P1 o que mostra que o problema P2 (Np-Completo) é mais fácil ou no máximo tão difícil quanto o problema P1. E como o problema P1 pode ser resolvido por um algoritmo determinista em tempo polinomial, P2 também poderia ser resolvido, mostrando que um problema NP-Completo pode ser resolvido em tempo polinomial por um algoritmo determinista e portanto, P = NP.

vi) Verdade. (Se a interseção entre P e NP-Completo, quer dizer que algum problema NP-Completo pode ser resolvido em tempo polinomial por um algoritmo determinista. E se apenas um problema NP-Completo puder ser resolvido dessa forma, sabe-se que qualquer problema NP-Completo também poderá ser resolvido em tempo polinomial por um algoritmo determinista, pois um problema NP-Completo é tão difícil quanto todos os outros problemas em NP-Completo, mas não mais difícil).