Nome: Daniel Brito (397824) | TEO - Lista 3

1) IDA) seja P(I) um problema de otimização (maximização), onde I representa uma instância de P, e P(I) tem como resposta o custo máximo de uma solução para I, ou uma indicação de que I é ilimitado, ou uma indicação de que I é inviável. Assuma também P(I,K), que consiste em decidir se a instância I tem uma solução de custo pelo menos K.

Suponha que existe um algoritmo polinomial para P(I). Note que, para retornar uma resposta, ele precisa, necessariamente, considerar todas as possibilidades possíveis. Assim, podemos torná-lo como parte da solução de P(I,K) da seguinte maneira: P(I,K) retorna indicando que há uma solução de custo pelo menos K para a instância I se, e somente se, a solução ótima de P(I) existir, e se ela for maior ou igual ao custo K. Uma vez que todas as possibilidades estão sendo consideradas, isto pode ser feito em tempo n^c, onde n é o tamanho da entrada e c é uma constante. Note que, se o custo K for menor do que a solução ótima de P(I), não existe uma solução de custo pelo menos K para a instância I.

Portanto, se existe um algoritmo polinomial para P(I), existe um algoritmo polinomial para P(I,K).

VOLTA) Se P(I,K) é resolvido em tempo polinomial, então, existe um algoritmo que resolve P(I,K) em tempo n^c, onde n é o tamanho da entrada e c é uma constante.

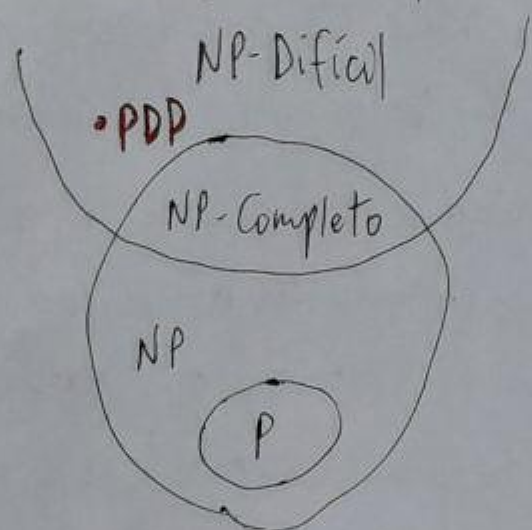
Assim podemos resolver P(I) da seguinte maneira: Executamos P(I,K) com K assumindo o menor valor possível. Então, enquanto P(I,K) for verdade, incrementamos o valor de K. Quando P(I,K) se tornar falso pela primeira vez, temos que K-1 é o valor máximo de P(I).

Portanto, uma vez que este teste é realizado, no máximo, n vezes, temos que o tempo total é dado por, no máximo, $n * n^c = n^{c+1}$. Ou seja, polinomial. ■

(2) a) Tome o Problema da Parada.

Como sabemos, o Problema da Parada é um problema de decisão clássico que consiste, basicamente, em determinar se um dado programa sempre vai terminar sua execução (parar) para uma dada entrada arbitrária, ou se vai executar infinitamente (entrar em loop).

Tal problema encontra-se na Classe NP-Difícil que, por definição, engloba os problemas que são tão difíceis quanto os problemas mais difíceis da Classe NP:



A classe NP, por sua vez, engloba o conjunto de problemas que são decidíveis em tempo polinomial por uma máquina de Turing não-determinística, ou seja, que são decidíveis em um número finito de operações.

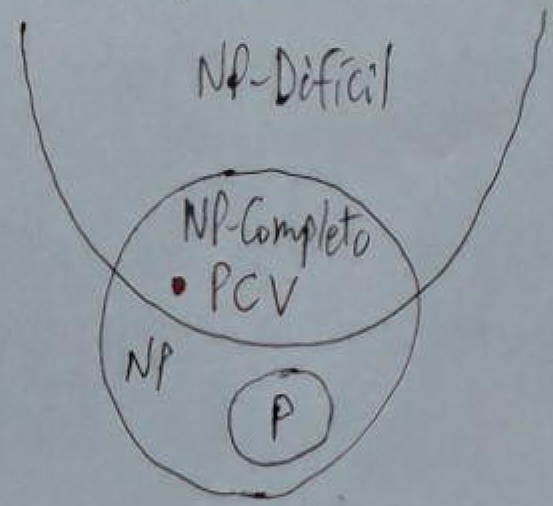
Portanto, como visto (e provado) no decorrer da disciplina, o Problema da Parada é indecidível. Logo, é um dos (possíveis) problemas que não se encontra em NP.

b) Tome o Problema do Caixeiro Viajante.

Como sabemos, o Problema do Caixeiro Viajante é um problema de decisão que, dado um conjunto de cidades, suas respectivas distâncias e uma constante K, é verificado se existe um roteiro entre tais cidades, de tal forma que o comprimento total do caminho seja menor ou igual à K.

A classe NP-Completo, por definição, engloba os problemas mais difíceis da Classe NP, sendo representados pela interseção entre as classes NP e NP-Difícil, os quais podem ser resolvidos por uma máquina de Turing de maneira não-determi rístico em tempo polinomial.

Em outras palavras, para um problema estar em NP-Completo, ele precisa estar, necessariamente, em NP e NP-Difícil:



Assim, é necessário verificar se o Problema do Caixeiro Viajante está de acordo com tais restrições.

Primeiramente, para provar que ele pertence à classe NP, pode-se realizar um passeio por cada cidade; em seguida, somar o custo total das distâncias; e, finalmente, verificar se o custo é mínimo. Algo que pode ser realizado em tempo polinomial e de maneira não-determinística. Logo, o Problema do Caixeiro Viajante pertence à classe NP.

Em segundo lugar, para provar que ele também pertence à classe NP-Difícil, pode-se realizar uma redução do Problema do Ciclo Hamiltoniano para o Problema do Caixeiro Viajante.

Portanto, uma vez que o problema do Caixeiro Viajante está tanto em NP quanto em NP-Difícil, temos que ele se encontra em $NP \setminus P$.

3) a) União:

Sejam as linguagens $L1, L2 \in P$. Assim, existem máquinas de Turing M1 e M2, que decidem, respectivamente, L1 e L2 em tempo polinomial. Ou seja, assuma que M1 possua tempo de execução $O(n^{k1})$, e que M2 possua tempo de execução $O(n^{k2})$, onde n é o tamanho da entrada w, e $k1$ e $k2$ são constantes.

Seja M3 uma máquina de Turing que decide $L1 \cup L2$. Desta forma, M3 se comporta da seguinte maneira:

Dada uma entrada w, $M1$ é executada para a entrada w; se $M1$ aceita, então, $M3$ aceita. $M2$, por sua vez, também é executada para a entrada w; se $M2$ aceita, então $M3$ aceita. Caso contrário, se ambas as máquinas $M1$ e $M2$ rejeitarem w, então, $M3$ também rejeita.

Em suma, $M3$ aceita w se, e somente se, $M1$ ou $M2$ (ou ambas) aceitam w. Assim, temos que o tempo de execução de $M3$, no pior caso, é dado por $O(n^{K1}) + O(n^{K2})$, o que ainda é polinomial.

Portanto, podemos concluir que $L1 \cup L2 \in P$. ■

ⓐ Interseção:

Sejam as linguagens $L1, L2 \in P$. Assim, existem máquinas de Turing $M1$ e $M2$, que decidem, respectivamente, $L1$ e $L2$ em tempo polinomial. Ou seja, assuma que $M1$ possua tempo de execução $O(n^{K1})$, e que $M2$ possua tempo de execução $O(n^{K2})$, onde n é o tamanho de uma entrada w, e $K1$ e $K2$ são constantes.

Seja $M3$ uma máquina de Turing que decide $L1 \cap L2$. Desta forma, $M3$ se comporta da seguinte maneira:

Dada uma entrada w, $M1$ é executada para a entrada w; se $M1$ aceita, então, $M2$ também é executada para a entrada w; se $M2$ aceita, então, $M3$ aceita. Note que estamos tratando de uma disjunção, logo, se $M1$ ou $M2$ rejeitarem a entrada w, consequentemente, $M3$ também rejeita.

Em suma, $M3$ aceita w se, e somente se, $M1$ e $M2$ aceitarem w. Assim, temos que o tempo de execução de $M3$, no pior caso, é dado por $O(n^{K1}) + O(n^{K2})$, o que ainda é polinomial.

Portanto, podemos concluir que $L1 \cap L2 \in P$. ■

b) União:

Sejam as linguagens $L1, L2 \in NP$. Assim, existem máquinas de Turing M1 e M2, que decidem, respectivamente, L1 e L2, de maneira não-determinística em tempo polinomial. Ou seja, assuma que M1 possua tempo de execução $O(n^{K1})$, e que M2 possua tempo de execução $O(n^{K2})$, onde n é o tamanho de uma entrada w, e K1 e K2 são constantes.

Neste contexto, o não-determinismo auxilia no processo por meio da execução de todos os testes de uma única vez, simultaneamente, através de múltiplas threads.

Assim, com base nesta ideia, seja M3 uma máquina de Turing que decide $L1 \cup L2$ em tempo polinomial e de maneira não-determinística. Desta forma, M3 se comporta da seguinte maneira:

Dada uma entrada w, uma thread de M3 executa, de maneira não-determinística, M1 para a entrada w; se M1 aceita, então, M3 aceita. Simultaneamente, através de outra thread de M3, e de maneira não-determinística, M2 é executada para a entrada w; se M2 aceita, então, M3 aceita. Como trata-se de uma conjunção, temos que M3 rejeita w apenas quando M1 e M2 rejeitarem w.

Em suma, M3 aceita w se, e somente se, M1 ou M2 (ou ambos) aceitarem w. Assim, temos que o tempo de execução de M3, no pior caso, é dado por $O(n^{K1}) + O(n^{K2})$, o que ainda é polinomial.

Portanto, podemos concluir que $L1 \cup L2 \in NP$. ■

b) Interseção:

Sejam as linguagens $L1, L2 \in NP$. Assim, existem máquinas de Turing M1 e M2, que decidem, respectivamente, L1 e L2, de maneira não-determinística em tempo polinomial. Ou seja, assuma que M1 possua tempo de execução $O(n^{K1})$, e que M2 possua tempo de execução $O(n^{K2})$, onde n é o tamanho de uma entrada w, e

K1 e K2 são constantes.

Neste contexto, o não-determinismo auxilia no processo por meio da execução de todos os testes de uma única vez, simultaneamente, através de múltiplos threads.

Assim, com base nesta ideia, seja M3 uma máquina de Turing que decide $L1 \cap L2$ em tempo polinomial, e de maneira não-determinística. Desta forma, M3 se comporta da seguinte maneira:

Dada uma entrada w, M1 e M2 são executados simultaneamente para a entrada w, por meio de diferentes threads de M3; se M1 aceita e M2 aceita, então, M3 aceita. Note que estamos tratando de uma disjunção, logo, se M1 ou M2 rejeitarem a entrada w, consequentemente, M3 também rejeita.

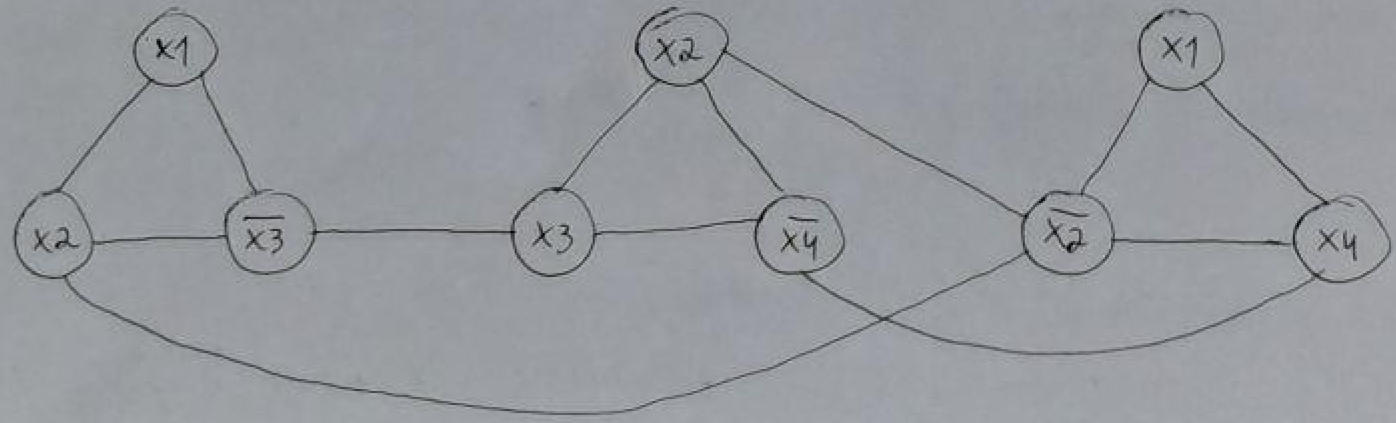
Em suma, M3 aceita w se, e somente se, M1 e M2 aceitam w. Assim, temos que o tempo de execução de M3, no pior caso, é dado por $O(n^{K1}) + O(n^{K2})$, o que ainda é polinomial.

Portanto, podemos concluir que $L1 \cap L2 \in NP$. ■

4) IDA) Para mostrar que o problema CONJ-INDEP é NP-Completo, assuma F como uma 3fnc-fórmula, tal qual:

$$(x1 \vee x2 \vee \overline{x3}) \wedge (x2 \vee x3 \vee \overline{x4}) \wedge (x1 \vee \overline{x2} \vee x4)$$

A redução de F gera um grafo não-direcionado G, que pode ser definido da seguinte forma:



(Observação: Este exemplo é apenas para fornecer um recurso visual para a demonstração, que está genérica.)

Note que, para cada cláusula, é criado um nó representando cada variável, possuindo como rótulo x_i. Além disso, para cada nó rotulado como x_i, é criado uma aresta que o conecta ao nó com rótulo representando sua negação $\overline{x_i}$.

A ideia consiste em provar que, se uma 3fnc-fórmula é satisfazível, então, G possui um conjunto independente de tamanho K.

Assuma que F seja satisfazível.

Como F é satisfazível, cada cláusula deve conter pelo menos um literal ao qual é atribuído o valor 1.

Agora, analisando a estrutura do grafo, o conjunto independente deve ter K nós, onde K representa o número de cláusulas.

Note que, cada cláusula corresponde a uma única tripla, e pode-se selecionar apenas um nós em cada cláusula, pois dois nós selecionados em cada cláusula não são independentes. Isto implica que podemos selecionar apenas um dos nós valorados com 1 em cada cláusula. Uma vez que isto ocorre, temos um conjunto independente.

Perceba também que x_i e $\overline{x_i}$ são contraditórios, não podendo ser selecionados ao mesmo tempo, uma vez que definimos apenas a seleção dos nós valorados como 1.

Baseado nisto, temos exatamente K nós que formam um conjunto independente.

Portanto, uma vez que F é satisfazível, temos que G possui um conjunto independente de tamanho K.

VOLTA) Seja G um grafo não-direcionado que possui um conjunto independente de tamanho K. Assim, existem K nós em G que não possuem arestas entre si. Assuma que estes K nós são valorados como 1.

Note que estes nós precisam estar em diferentes triplas, e que os nós que formam estas triplas estão conectados. Além disto, para cada nó rotulado como x_i

é criado uma aresta que o conecta ao nó com rótulo representando sua negação $\overline{x_i}$

Seja F uma 3fnc-fórmula.

Desta maneira, cada tripla de G pode corresponder a uma cláusula de F. Para que F seja satisfazível, cada cláusula deve conter pelo menos um literal ao qual é atribuído o valor 1, o que garantimos ao definir o valor 1 aos K nós que formam o conjunto independente.

Temos ainda que as variáveis com rótulos contraditórios, como x_i e $\overline{x_i}$, não podem estar na mesma cláusula, uma vez que definimos a seleção dos nós valorados como 1.

Portanto, temos que G é consistente com todas as cláusulas de uma 3fnc-fórmula satisfazível. ■

5) Suponha a existência de uma subrotina CONJ-INDEP (G,K) que, dado um grafo G, verifica se o mesmo possui um conjunto independente de tamanho K. Caso positivo, CONJ-INDEP (G,K) retorna 1, caso contrário, retorna 0.

Uma vez que estamos interessados em encontrar o conjunto independente máximo, podemos utilizar uma subrotina auxiliar, por meio de CONJ-INDEP, para verificar o tamanho do maior conjunto independente em G, conforme a seguir:

```
TAM-CONJ-INDEP-MAX (G):
    para K de 1 até |V|:
        se CONJ-INDEP (G, K+1) == 0 :
            retorne K
    retorne 0
```

Temos que um conjunto de vértices S é considerado independente em um grafo G, se não existe aresta para quaisquer par de vértices de S no grafo G.

A fim de construir um algoritmo para encontrar o conjunto independente máximo, podemos utilizar a subrotina TAM-CONJ-INDEP-MAX(G). Para tal, assuma G - v como sendo um grafo no qual o vértice v é desconsiderado. Baseado nisto, podemos construir o seguinte algoritmo:

```
CONJ-INDEP-MAX(G):
    S = { }
    K = TAM-CONJ-INDEP-MAX(G)

    para todo v em G:
        se TAM-CONJ-INDEP-MAX(G-v) ≠ K:
            adicione v em S

    retorne S
```

A ideia geral consiste em, primeiramente, obter o tamanho do conjunto independente máximo, e utilizar este valor na verificação de um conjunto independente máximo de G, mas "excluindo" v durante o processo de iteração.

Neste caso, é considerado a existência de um único conjunto independente máximo, pois note que a "remoção" de um elemento que deve pertencer ao conjunto independente máximo torna a condição TAM-CONJ-INDEP-MAX(G-v) ≠ K verdadeira, ou seja, tal elemento deve compor o conjunto, de maneira que o mesmo tenha tamanho igual a K. Perceba que os outros vértices não se enquadram nesta situação.

Analisando o desempenho deste algoritmo, temos que um grafo G com n vértices não pode ter um conjunto independente maior que n, logo, TAM-CONJ-INDEP-MAX sempre retorna um valor. Além disto, se G possui um conjunto independente de tamanho K, consequentemente, terá um conjunto de tamanho K-1.

Desta forma, se CONJ-INDEP possui tempo de execução $O((|V|+|E|)^c)$, então, TAM-CONJ-INDEP-MAX possui tempo de execução $O((|V|+|E|)^{c+1})$, ou seja, ainda polinomial.

Portanto, temos que o tempo de execução de CONJ-INDEP-MAX é polinomial em $|V|$ e $|E|$.

6 Seja o problema SET-PARTITION, que recebe como entrada um conjunto X de números, e verifica se eles podem ser particionados em dois outros subconjuntos, tal como:

$$\sum_{x \in A} x = \sum_{x \in \bar{A}} x$$

Assuma que o conjunto X possua duas partições, A e $\bar{A} = X - A$. Assim, devemos verificar se cada partição possui elementos que, uma vez somados, possuem valor igual.

Neste caso, iremos fazer uma redução de SUBSET-SUM para SET-PARTITION.

O SUBSET-SUM é definido da seguinte maneira: Dado um conjunto X de inteiros e um alvo t, é verificado se existe um subconjunto $Y \subseteq X$, tal que os elementos de Y somados resultam em t.

Seja s a soma dos valores de X. Além disto, assuma $X' = X \cup \{s-2t\}$ como sendo uma entrada para SET-PARTITION.

Desta forma, podemos fazer com que SUBSET-SUM retorne verdade se, e somente se, SET-PARTITION retornar verdade. Em outras palavras, devemos provar que uma instância $(X,t) \in$ SUBSET-SUM se, e somente se, a instância $(X') \in$ SET-PARTITION. Note que, pela definição anterior, temos que a soma dos elementos de X' resulta em $2s-2t$.

Perceba também que esta redução pode ser feita em tempo polinomial, uma vez que pode-se simplesmente somar os valores do conjunto de maneira não-determinística. Assim, com base nesta ideia, temos que:

IDA) Se existe um conjunto de números X cuja soma resulta em t, então, os elementos restantes em X somados resulta em $s-t$.

Portanto, X' pode ser particionado em dois subconjuntos cuja soma dos elementos de cada um resulte em $s-t$:

$$X' = \underbrace{X}_{t \qquad s-t} \cup \{s-2t\}$$

+ (Somado)

$$\downarrow$$

$$s-t \cup s-t$$

VOLTA) Suponha que X' possa ser particionado em dois subconjuntos, de tal forma que a soma dos elementos de cada um resulte em $s-t$.

Note que, pela definição anterior, sabemos que um destes subconjuntos contém o número $s-2t$, e que a soma dos valores de X é s. Com isto, temos que a soma dos elementos de X' resulta em $2s-2t = s+s-t-t$, podendo ser representada por $s-t \cup s-t$.

Portanto, uma vez que a soma de tais elementos pode ser realizada em tempo polinomial e de maneira não-determinística, temos que X' pode ser particionado em dois subconjuntos cujo soma dos elementos de cada um resulta em $s-t$. ■