Universidade Federal de Roraima

Departamento de Ciência da Computação

DISCIPLINA: DCC606

LISTA 2 - Prazo de Entrega: 30/05/2019 até às 23:55h

ALUNO(A): Larissa Santos Silva NOTA:_____

ATENÇÃO: Descrever as soluções com o máximo de detalhes possível, no caso de programas (escritos em C ou C++), inclusive a forma como os testes foram feitos. Todos os artefatos (relatório, código fonte de programas, e outros) gerados para este trabalho devem ser adicionados em um repositório no site github.com com os seus pontos extras da disciplina.

1) Descreva o que é a NP-Completude.

Pertence à classe NP-Completo se: primeiro, pertence à classe NP; e em segundo, todos os problemas da classe NP puderem ser redutíveis a ele. Os problemas NP-Completos tem a interessante propriedade de que, se um problema NP-Completo tiver solução determinística polinomial, qualquer outro problema NP também tem.

Esta propriedade é posta a partir do conceito de redução polinomial. O fato de não estar provado que P=NP e acreditar-se que P ≠ NP torna a identificação de um problema na classe NP-Completa uma questão na prática muito importante, pois determina o não conhecimento de solução eficiente para o problema. Há problemas que estão numa classe “meio nebulosa” como o de decidir se duas expressões regulares são equivalentes e o de decidir se uma palavra é gerada por uma certa gramática sensível ao contexto. Esses problemas são polinomialmente reduzíveis de problemas NP-Completos, entretanto não se conhece algoritmo não determinístico de tempo polinomial que os resolva, e portanto não se sabe se pertencem ou não a P. Esses problemas são chamados NP-Difíceis.

2) Apresente 5 problemas provados ser NP-Completo, com suas respectivas referências.

- Fatoração e um NP-completo, porque um número sempre pode ser fatorado em números primos.

- Código em C#

```
#include <stdio.h>
#include <stdlib.h>

int fatorial(int n, int fat);

main(){
   int n, fat=1;

   scanf("%i",&n);
   fatorial(n,fat);
   getch ();
```

```
   return (0);
}

int fatorial(int n, int fat){
   if (n==0){
         printf("fatorial = %i",fat);
   return (0);
   }

   fat = fat * n;
   fatorial(n-1, fat);
}
```

- Torre de Hanói e um problema exponenciais para os quais se comprovou que tanto achar quanto verificar só ocorrem em tempo exponencial.

- Código em C#

```
#include <stdio.h>
#include <stdlib.h>

int torre(int disco, char origem, char destino, char aux);

int main(){

    int n_discos;
    printf("\nDigite o numero de discos: ");
    scanf("%d",&n_discos);
    torre(n_discos,'A','B','C');
    return 0;
}

int torre(int disco, char origem, char destino, char aux){
   if(disco == 1){
         printf("Mover o disco %d da torre %c para a torre %c\n",disco, origem, destino);
   }else{
      torre(disco-1, origem, aux, destino);
      printf("Mover o disco %d da torre %c para a torre %c\n",disco, origem, destino);
      torre(disco-1,aux, destino, origem);
   }
}
```

-Caixeiro viajante e um NP-completo, porque existe um ciclo que percorra todos os vértices do grafo com um custo $\leq$ B.

- Código em C#

```
#include <stdio.h>
#include <stdlib.h>

//site de referência: https://sites.google.com/site/onildoribeiro/produo-tcnica/caixeiro-viajante

int calcular();

main(){
   char r;
   while(1){
```

```
        printf("\n\n\t\t\t____Caixeiro Viajante____\n\n");
        printf("\n\n\nDigite uma das opcoes:\n\n");
        printf("1 - Calcular a menor distancia e a Rota.\n");
        printf("2 - Sair\n\n");
        r = getchar();

        switch(r){
           case '1': calcular();
           break;

           case '2': exit(0);
        }
   }
}
int calcular(){
   int mtrx [4][4]={0,9,4,8,9,0,5,8,4,5,0,3,8,8,3,0};
   int i,j,k,m,n,r,s,o,c=0,x;
   int v[6]={0};
   int vc2[6]={0};
   int vc3[6]={0};
   int vc4[6]={0};

   printf("\n   Cidade 1\n   Cidade 2\n   Cidade 3\n   Cidade 4\n\n");
   printf("Digite o numero a cidade de Origem:  ");

   scanf("%d",&x);

   if((x<1)||(x>4)){
        printf("\n\nCidade errada");
   getch();
   }
   else{
      j=i=(x-1);
      v[c]=mtrx[j][i];

      for(k=0;k<4;k++){
           if(k!=i){
              v[c]=v[c]+mtrx[j][k];
              v[c+1]=v[c];
              vc2[c]=k+1;
              vc2[c+1]=vc2[c];

              for(m=0;m<4;m++){
                   for(n=0;n<4;n++){
                      if((n==k)&&(m!=j)&&(m!=k)){
                         v[c]=v[c]+mtrx[m][n];
                          vc3[c]=m+1;

                          for(r=0;r<4;r++){
                               for(s=0;s<4;s++){
                                  if((r==m)&&(s!=i)&&(s!=n)&&(s!=r)){
                                     v[c]=v[c]+mtrx[r][s];
                                     vc4[c]=s+1;
                                     c++;
                                     }
                                  }
                               }
                            }
```

```
                    }
                  }
                }
              }

              for(i=0;i<6;i++){
                    if(v[i]<v[i+1]){
                       o=i;
              if(v[5]<v[i])
                 o=5;
                    }
              }

              getch();

              for(i=0;i<6;i++)
              printf("[%d]\n",v[i]);
              printf("\nSaindo da cidade %d: o menor caminho seria passando\npela cidade %d depois por %d e por
fim em  %d. \n\nNo total seria %d Km.\n",x,vc2[o],vc3[o],vc4[o],v[o]);
              getch();
        }
     return 0;
}
```

3) Apresente um lauda sobre o artigo:

S. Cook, The complexity of theorem-proving procedures, Proceedings of the 3rd Symposium on the Theory of Computing, ACM, pp.151-158. 1971.

Disponivel em: https://dl.acm.org/citation.cfm?coll=GUIDE&dl=GUIDE&id=805047The complexity of theorem-proving procedures