Documentação Trabalho Prático III

Algoritmos I

Nome: Matheus Schimieguel Silva
Matrícula:2017014413

1.Introdução

1.1 Descrição do problema

O problema consiste em achar a menor quantidade de bandeiras distribuídas entre os pontos de modo que todos os caminhos incidam em pelo menos um ponto .De modo que todo caminho conecta dois pontos e todo ponto tem pelo menos um caminho ao qual é conectado de maneira que não exista mais de um caminho que ligue diretamente dois pontos e nem um caminho circular que ligue um ponto nele mesmo.Assim, para a sua solução o problema será representado como um grafo G=(V,E) simples não direcionado em que as arestas são os caminhos e os vértices são os pontos.

A descrição do problema é mesma do problema np-hard minum vertex cover que possui tempo polinomial para achar a solução exata em grafos árvores como pedido na tarefa 1 e tempo exponencial para a solução exata em grafos que contenham ciclos como pedido na tarefa 2.

2. Estrutura de Dados e Algoritmos Utilizados

2.2 Estrutura de Dados

A estrutura de dados escolhida para representar o grafo G=(V,E) tanto na tarefa 1 quanto na tarefa 2 foi a lista de adjacência.A lista de adjacência é formada por um vetor de ponteiro de lista de tamanho '|V|'.De forma que cada lista representa um vértice e os seus elementos representam os vértices ao qual este é ligado .

As listas são composta por dois ponteiros para node, head e tail e inteiro que guarda o tamanho. Cada node tem um inteiro e um ponteiro para o próximo node.

Uma aresta qualquer e = (a,b) se a < b é representada como o número 'b' presente na lista de posição 'a' no vetor de listas por outro lado se b > a então a aresta 'e' será representada como o número 'a' presente na lista de posição 'b'.Assim ,cada aresta é representada apenas uma vez de modo que a primeira lista sempre terá um elemento e a lista referente ao último vértice nunca terá algum elemento.

2.3 Algoritmos

2.3.1 Tarefa 1

Foi pedido para se achar a solução exata em até 2 segundos em um grafo que não continha ciclos e fosse conexo ,isto é ,uma árvore . Assim foi escolhido um algoritmo de programação dinâmica com tempo de execução polinomial recursivo com o seguinte pseudo-código :

seja :

```
listaAdj a lista de adjacência
parent um array de dimensão n que guarda o pai de um no
visitedNodes um array de dimensão nX2 que que armazena
cálculos de custo para que não seja preciso recalculá-los
```

```
Function Vertex(atualnode,isVertexCover):
      if listaAdj[atualnode].tamanho == 0
            Return isVertexCover
      else if visitedNodes[atualnode][isVertexCover] != -1
      //resultado armazenado
          Return visitedNodes[atualnode][isVertexCover]
      end if
      soma := 0
      for i in range (listaAdj[atualnode].tamanho)
          element := listaAdj[atualnode][i]
          if element != parent[atualnode]
              parent[element] := atualnode
              if isVertexCover == 0
                  sum := sum + Vertex(element,1)
              else
                   sum := sum + min(Vertex(element,1), Vertex(element,0)
              end if
          end if
      end for
      visitedNodes[atualnode][isVertexCover] := sum + isVertexCover
      Return visitedNodes[atualnode][isVertexCover]
```

ao se executar min(Vertex(0,1),Vertex(1,1) tem-se o vertex cover de menor custo em uma árvore qualquer.

O algoritmo percorre a lista de adjacência partindo sempre do vértice 0 e indo em direção aos filhos de modo a passar um única vez por no.

A corretude do algoritmo se deve a caso o nó avaliado atual não estiver no vertex cover analisado então o seu filho precisa estar mas se o nó atual estiver no vertex então deve-se considerar os menores custos de se adicionar um filho ou não no vertex cover.Por isso sendo preciso considerar o caso de a raiz pertencer ou não ao vertex cover.

2.3.2 Tarefa 2

Foi pedido uma heurística com solução até duas vezes pior que a ótima com até 2 segundos de execução desse modo a heurística escolhida foi a de seguinte pseudo-código:

```
while |E| > 0
        seja e = (u,t) // uma aresta arbitrária de G = (V,E)
        adicionar 'u' e 't' no vertex cover.
        remover todas as arestas que passam por 'u' ou 't'.
```

O pior caso dessa heurística é quando tem-se um grafo em linha reta pois nesse caso sempre ao escolher uma aresta será eliminado no máximo duas arestas e será preciso uma maior quantidade de vértices escolhidos .Nesse caso o minum vertex cover é igual à metade da quantidade de vértices (|E|) e a heurística retorna a |V|.Portanto a heurística ,em seu pior caso, tem resultado duas vezes pior que a solução ótima.

3.Análise De Complexidade

seja ‘n’ o número de vértices e ‘m’ o número de arestas .

3.1 Espaço

Tanto a tarefa 1 e tarefa 2 usam como estrutura de dados a lista de adjacência .Por definição na lista de adjacência cada aresta é representada como um elemento na lista se e somente se ela estiver no grafo de maneira que toda aresta que não está presente no grafo não está presente na lista de adjacência.

Assim pela própria definição da lista de adjacência tem-se que o espaço ocupado em razão da lista de adjacência tanto pela tarefa 1 quanto pela tarefa 2 é O (m).

Na tarefa 1 são usado dois vetores em razão do algoritmo de programação dinâmica. Um de dimensão n x 1 e o outro de dimensão n X 2 portanto ocupando espaço O(n) como se trata de grafos árvores pode-se dizer que O(n) = O(m).

Na tarefa 2 é usado um vetor de dimensão n X 1 para guardar os vértices que pertence ao vertex cover.Como o grafo conexo que pode conter ciclos então O(n) = O (m)

Assim a complexidade total de espaço para tarefa 1 e tarefa 2 é O(m).

3.2 Tempo

3.2.1 Tarefa 1

No pseudocódigo presente em **2.3.1** percebe-se que que o algoritmo percorre de país em direção aos filhos uma quantidade constante e armazenando os cálculos em dois arrays de suporte .Dessa forma , o custo de tempo do algoritmo é O(m) .

3.2.2 Tarefa 2

A analisando-se o pseudocódigo pela sua simplicidade pode se estimar que a sua complexidade seja linear em relação ao número ‘m’ de arestas por se usar uma lista de adjacência como estrutura de dados.

4.Avaliação Experimental

Para auxiliar a anaĺise e os testes a avaliação experimental foi feita em python no jupyter notebooks onde é possível facilmente gerar várias entradas e plotar gráficos das performances das médias e do desvio padrão para uma grande quantidade de testes que seriam inviáveis de serem feitos manualmente.

Para a avaliação experimental foram criadas entradas para dois tipos de árvores uma chamada de linetree(em linha) e o outro tipo de árvore foi uma árvore binária .Foram gerados cerca de 200 arquivos de texto para cada tipo de árvore que possuem até 10**5 vértices.

Por fim foi feito um teste com grafos completos para testar a heurística .

4.1 geradores de entrada

as entradas geradas e usada neste trabalho para a avaliação experimental se encontram na pasta avaliação experimental/entradas.

In [1]:
```
import numpy as np
import os
import time
import matplotlib.pyplot as plt
import math
from subprocess import check_output
```

In [2]:
```
n1 = np.arange(2,10**5,500)
n2 = np.arange(3,10**5,512)
n3 = np.arange(3,2000,20)
tarefas = ["tarefa1","tarefa2"]
n1.shape
```

Out[2]:
```
(200,)
```

In [3]:
```
#gerando arquivos txt arvores linha reta
if not os.path.exists("entradas"):
    os.mkdir("entradas")
if not os.path.exists("entradas/linetree"):
    os.mkdir("entradas/linetree")
for i in range(len(n1)):
    if not os.path.isfile("entradas/linetree/t"+str(i)+".txt"):
        with open("entradas/linetree/t"+str(i)+".txt","w") as f:
            f.write(str(n1[i]) +" "+ str(n1[i]-1)+"\n")
            for j in range(n1[i]-1):
                f.write(str(j)+ " " +str(j+1)+"\n")
```

In [4]:
```
#arvores arquivos txt arvores binarias
if not os.path.exists("entradas"):
    os.mkdir("entradas")
if not os.path.exists("entradas/binarytree"):
    os.mkdir("entradas/binarytree")
for i in range(len(n2)):
    if not os.path.isfile("entradas/binarytree/t"+str(i)+".txt"):
        with open("entradas/binarytree/t"+str(i)+".txt","w") as f:
            f.write(str(n2[i]) +" "+ str(n2[i]-1)+"\n")
            l = 1
            for j in range(math.floor(n2[i]/2)-1):
                for k in range(2):
                    f.write(str(j) +" "+ str(l)+"\n")
                    l = l + 1
```

4.2 árvores em linha

In [1]:
```
import numpy as np
import os
import time
import matplotlib.pyplot as plt
import math
from subprocess import check_output
```

In [2]:
```
n1 = np.arange(2,10**5,500)
tarefas = ["tarefa1","tarefa2"]
n1.shape
```

Out[2]:
```
(200,)
```

In [3]:
```
#contando os tempos de execução
qtd = 10
tempos = np.zeros((2,n1.shape[0], qtd))
for t in range(len(tarefas)):
    for j in range(len(n1)):
        for i in range(qtd):
            #marcando o tempo de execução
            inicio = time.time()
            os.system('./../bin/tp3 '+ tarefas[t] +' "entradas/linetree/t'+str(j)+'.txt"')
            fim = time.time()
            tempos[t,j,i] = fim - inicio
```

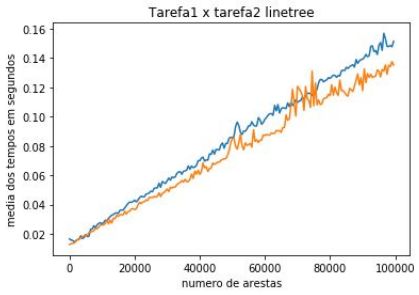
Média do tempo do tempo de execução para tarefa 1 em azul e tarefa 2 em laranja (para cada entrada o algoritmo foi executado 10 vezes).

In [4]:
```
#media dos tempos de execuçao
mediaTempoTarefa1 = np.mean(tempos[0],axis=1)
mediaTempoTarefa2 = np.mean(tempos[1],axis=1)
#desvio padrao dos tempos
DesvioPadraoTarefa1 = np.std(tempos[0],axis=1)
DesvioPadraoTarefa2 = np.std(tempos[1],axis=1)
```

In [6]:
```
#plotando as medias dos tempos de excuçao
plt.title("Tarefa1 x tarefa2 linetree")
plt.ylabel('media dos tempos em segundos')
plt.xlabel('numero de arestas')

plt.plot(n1,mediaTempoTarefa1)
plt.plot(n1,mediaTempoTarefa2)
```

Out[6]:
```
[<matplotlib.lines.Line2D at 0x7fb33a0e4e80>]
```

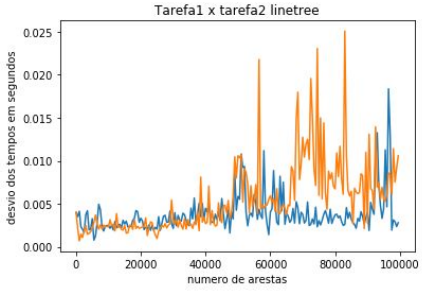


Desvio Padrão do tempo de execução para tarefa 1 em azul e tarefa 2 em laranja (para cada entrada o algoritmo foi executado 10 vezes).

In [8]:
```
#plotando os desvios padrao dos tempos de execuçao
plt.title("Tarefa1 x tarefa2 linetree")
plt.ylabel('desvio dos tempos em segundos')
plt.xlabel('numero de arestas')

plt.plot(n1,DesvioPadraoTarefa1)
plt.plot(n1,DesvioPadraoTarefa2)
```

Out[8]:
```
[<matplotlib.lines.Line2D at 0x7fb339fb6908>]
```



In [9]:
```
#comparando solucao heuristica com solucao polinomial
#line tree
result = np.zeros((n1.shape[0]))
tarefa1 = np.zeros((n1.shape[0]))
tarefa2 = np.zeros((n1.shape[0]))
for j in range(len(n1)):
    tarefa1[j] = int (check_output('./../bin/tp3 "tarefa1" "entradas/linetree/t'+str(j)+'.txt"', shell=True))
    tarefa2[j] = int (check_output('./../bin/tp3 "tarefa2" "entradas/linetree/t'+str(j)+'.txt"', shell=True))
    result[j]  = tarefa2[j]/tarefa1[j]
```

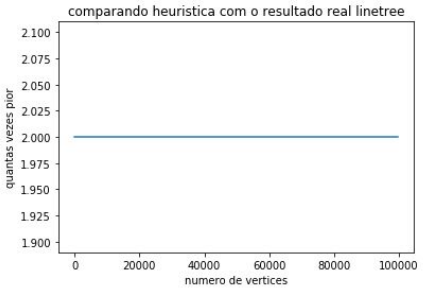
In [10]:
```
#plotando as medias dos tempos de excuçao
plt.title("comparando heuristica com o resultado real -linetree")
plt.ylabel('quantas vezes pior')
plt.xlabel('numero de vertices')

plt.plot(n1,result)
```

Gráfico correlacionando a razão entre a resposta dada pela heurística (tarefa 2) e a resposta dada pelo algoritmo de programação dinâmica(tarefa 1).

Perceba que o gráfico é exatamente uma função constante de valor 2 que a pior razão de valor de retorno esperado para a heurística exatamente no pior caso imaginado para ela.

```
Out[10]: [<matplotlib.lines.Line2D at 0x7fb339f74fd0>]
```



4.3 árvores Binárias

```
In [1]: import numpy as np
        import os
        import time
        import matplotlib.pyplot as plt
        import math
        from subprocess import check_output

In [2]: n2 = np.arange(3,10**5,512)
        tarefas = ["tarefa1","tarefa2"]
        n2.shape

Out[2]: (196,)

In [3]: #contando os tempos de execução
        qtd = 10
        tempos2 = np.zeros((2,n2.shape[0], qtd))
        for t in range(len(tarefas)):
            for j in range(1,len(n2)):
                for i in range(qtd):
                    #marcando o tempo de execução
                    inicio = time.time()
                    os.system('./../bin/tp3 '+ tarefas[t] +' "entradas/binarytree/t'+str(j)+'.txt"')
                    fim = time.time()
                    tempos2[t,j,i] = fim - inicio
```

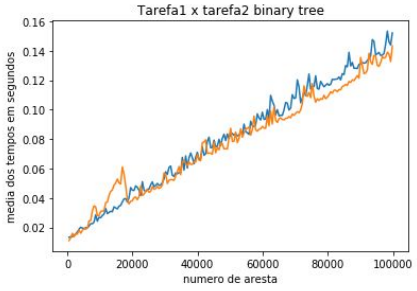
Média do tempo do tempo de execução para tarefa 1 em azul e tarefa 2 em laranja (para cada entrada o algoritmo foi executado 10 vezes).

In [4]:
```
#media dos tempos de execuçao
mediaTempoTarefa1 = np.mean(tempos2[0],axis=1)
mediaTempoTarefa2 = np.mean(tempos2[1],axis=1)
#desvio padrao dos tempos
DesvioPadraoTarefa1 = np.std(tempos2[0],axis=1)
DesvioPadraoTarefa2 = np.std(tempos2[1],axis=1)
```

In [8]:
```
#plotando as medias dos tempos de excuçao
plt.title("Tarefa1 x tarefa2 binary tree")
plt.ylabel('media dos tempos em segundos')
plt.xlabel('numero de aresta')

plt.plot(n2[1:],mediaTempoTarefa1[1:])
plt.plot(n2[1:],mediaTempoTarefa2[1:])
```

Out[8]: `[<matplotlib.lines.Line2D at 0x7f2a1fc2a860>]`



In [6]:
```
#comparando solucao heuristica com solucao polinomial
#binary tree

result = np.zeros((n2.shape[0]))
tarefa1 = np.zeros((n2.shape[0]))
tarefa2 = np.zeros((n2.shape[0]))
for j in range(1,len(n2)):
    tarefa1[j] = int (check_output('./../bin/tp3 "tarefa1" "entradas/binarytree/t'+str(j)+'.txt"', shell=True))
    tarefa2[j] = int (check_output('./../bin/tp3 "tarefa2" "entradas/binarytree/t'+str(j)+'.txt"', shell=True))
    result[j]  = tarefa2[j]/tarefa1[j]
```

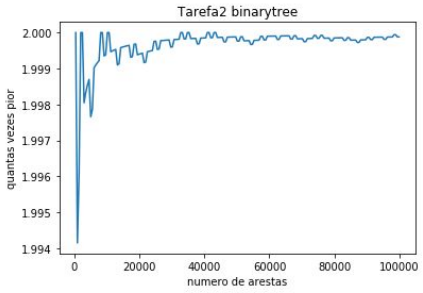
In [9]:
```
#plotando as medias dos tempos de excuçao
plt.title("Tarefa2 binarytree")
plt.ylabel('quantas vezes pior')
plt.xlabel('numero de arestas')

plt.plot(n2[1:],result[1:])
```

Gráfico correlacionando a razão entre a resposta dada pela heurística (tarefa 2) e a resposta dada pelo algoritmo de programação dinâmica(tarefa 1).

Perceba que o gráfico tem como teto justamente o valor 2 que a pior razão de valor de retorno esperado para a heurística.

Out[9]: `[<matplotlib.lines.Line2D at 0x7f2a1fc00a90>]`

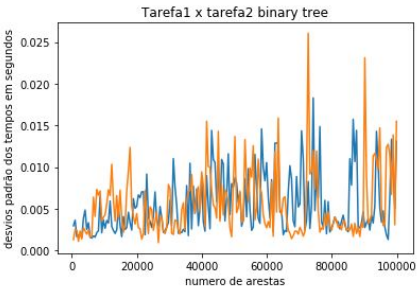


In [11]:
```
#plotando os desvios padrão dos tempos de excuçao
plt.title("Tarefa1 x tarefa2 binary tree")
plt.ylabel('desvios padrão dos tempos em segundos')
plt.xlabel('numero de arestas')

plt.plot(n2[1:],DesvioPadraoTarefa1[1:])
plt.plot(n2[1:],DesvioPadraoTarefa2[1:])
```

Desvio Padrão do tempo do tempo de execução para tarefa 1 em azul e tarefa 2 em laranja (para cada entrada o algoritmo foi executado 10 vezes).

```
Out[11]: [<matplotlib.lines.Line2D at 0x7f2a1fad4198>]
```



4.4 Gráfico com ciclos

Foram gerados 11 arquivos de texto que representam grafos completos de 3 a 15 vértices.Para grafos completos o valor de minimum vertex cover é sempre ‘n - 1’ vértices.

A heurística por sua vez retorna um vertex cover de tamanho ‘n’ se ‘n’ for par do contrário retorna ‘n-1’.Com isso para um K3 a heurística retorna um resultado 35 % pior de modo que apesar de oscilar para um Kn com n > 3 a heurística tende aproximar cada vez mais do resultado ótimo.

```
In [14]: n3 = np.arange(3,15)
         result = np.zeros((n3.shape[0]))
         tarefa1 = np.zeros((n3.shape[0]))
         tarefa2 = np.zeros((n3.shape[0]))
         for j in range(len(n3)):
             tarefa2[j] = int (check_output('./../bin/tp3 "tarefa2" "entradas/kgraph/k'+str(j)+'.txt"', shell=True))
             result[j]  = tarefa2[j]/(n3[j]-1)
```

```
In [18]: #plotando as medias dos tempos de excuçao
         plt.title("Tarefa 2 para k grafos")
         plt.ylabel('quantas vezes pior')
         plt.xlabel('numero de vertices')

         plt.plot(n3,result)
         print (tarefa2)
```

```
[ 2.  4.  4.  6.  6.  8.  8. 10. 10. 12. 12. 14.]
```