# Trabalho 1
# de
# Implementação

# Análise de Algoritmos

# 2011.1

## Relatório

Autores:
Carla Galdino - 0713101
Dario Andrade Tinoco de Souza - 9814309
Eliana Goldner - 0712600

O trabalho foi implementado com C++ e a máquina utilizada foi um MacBook Pro Core 2 Duo 2.8 GHz, 8 GB de memória, MacOS 10.6.

Para o trabalho foi criada uma classe AdjacencyList que representa o grafo e define alguns métodos (RemoveHighestDegreeVertex, UpdateData, etc) que devem ser implementados por cada classe derivada dela. Essa classe possui os seguintes atributos:

- nVertex - números de vértices
- arrAdjList - lista de adjacências de cada vértice do grafo
- nEdges - número de arestas

Para cada tarefa, foi criada uma classe que extende AdjacencyList e implementa detalhes do algoritmo proposto. A seguir especificamos atributos e métodos de cada implementação através de um pseudo-código.

A classe AlgoritmoCoberturaGulosa possui uma função básica genérica para implementar a estratégia gulosa:

```
while ( grafo.HasEdge )

      vertex ← grafo.RemoveHighestDegreeVertex();

      listaCobertura ← listaCobertura ∪ vertex;

end while
```

**Código fonte:**

Use o repositório Subversion (svn) para baixar o código do projeto:

svn checkout ***http***://aa-trabalho1-20111.googlecode.com/svn/trunk/ aa-trabalho1-20111-read-only

**Tarefa 3:**

Para essa tarefa foi implementada a classe DegreeVectorAdjacencyList que mantem um vetor ( vectorDegrees ) de tamanho nVertex, o qual guarda o grau de cada vértice. Para descobrir o vértice de maior grau percorre-se esse vetor. A implementação do algoritmo é descrita a seguir:

**Classe DegreeVectorAdjacencyList:**

```
RemoveHighestDegreeVertex()
	vertex ← GetHighestDegreeVertex()
	for ( i ← vizinho de vertex )
		# se o grau desse vertice for 0, ele ja foi processado e removido do grafo
		# mas permanece na lista de adjacencias por questao de performance

		if d(vertex) > 0
			nEdges ← nEdges - 1
			DecrementDegree( i )
		end if
	end for
	d(vertex) ← 0
	return vertex;
end


GetHighestDegreeVertex()
	highestDegree ← 0
	vertex ← -1
	for ( i ← 1 .. nVertex )
		if ( d(i) > highestDegree )
			highestDegree  ← d(i)
			vertex ← i
		end if
	end for

	return vertex
end


DecrementDegree( i )
	vectorDegrees[i] ← vectorDegrees[i] - 1;
end
```

**Tarefa 4:**

Para essa tarefa foi implementada a classe DegreeHeapAdjacencyList que mantem uma maxHeap (heap) preenchida com elementos que são pares <vertexIndex, vertexDegree>. O grau do vértice é usado como critério para posicionar um nó da heap, de modo que o par com o maior grau está na raiz da heap. Para descobrir o vértice de maior grau remove-se a raiz da heap. Além disso, a heap mantem um vetor (indexerHeap) de tamanho nVertex, que guarda para cada vértice sua posição atual na heap. Desse modo, para atualizar o grau de um nó, basta acessá-lo diretamente. Outro atributo da heap é nextAvailableSlot que marca onde será inserido o próximo elemento.

**Classe DegreeHeapAdjacencyList:**

```
RemoveHighestDegreeVertex()

        [ vertex, degree ]  ← heap.RemoveRoot()
        for ( i ← vizinho de vertex )
                # ele pode ser vizinho mas nao pertencer mais ao heap
                # caso ele ja tenha sido removido e processado anteriormente
                # vide observacao tarefa 3

                if ( i ainda pertence ao heap )
                        nEdges ← nEdges - 1
                        heap.DecrementDegree( i )
                end if
        end

        return vertex
end
```

**Classe Heap:**
Foi implementada utilizando um vetor (heapVector).

```
Insert(vertex, degree)

        heapVector[ nextAvailableSlot ] ← [ vertex, degree]
        indexerHeap[ vertex ] ←  nextAvailableSlot
        nextAvailableSlot ← nextAvailableSlot + 1

        BubleUpElement( nextAvailableSlot - 1)
end
```

```
RemoveRoot()

        root ← heapVector[ 0 ]

        [ newRoot, newRootDegree ] ←  heapVector[ nextAvailableSlot - 1 ]
        heapVector[ 0 ] ← [ newRoot, newRootDegree ]

        bubleDownElement( 0 )

        return root
end

DecrementDegree( vertex )
        posicaoNoHeap ← indexerHeap[vertex ]
        heapVector[posicaoNoHeap].degree  ← heapVector[posicaoNoHeap].degree - 1

        bubleDownElement(posicaoNoHeap)
end

BubleUpElement( index )
        if ( index = 0 ) return

        swapped ← swapWithFather( index )
        if ( swapped )
                BubleUpElement(( index  - 1) / 2)
        end if
end

SwapWithFather( index )

        vertex ← heapVector[ index ]
        father ← heapVector[ (index-1) / 2 ]

        if ( vertexDegree > fatherDegree )

                 heapVector[(index - 1) / 2] ← vertex
                 heapVector[index] ← father

                 indexerHeap[ vertexIndex] ← (index - 1) / 2
                 indexerHeap[fatherIndex ] ← index;
```

```
            return true
        end  if
        return false
end

BubleDownElement( index )

        if ( index >= nextAvailableSlot) return

        swapped ← SwapWithChildren( index )

         if ( swapped = leftChild )
                BubleDownElement( 2 * index + 1)
        else if ( swapped == rightChild )
                BubleDownElement( 2 * index + 2 )
        end if
end

SwapWithChildren( iSlotIndex )

        node ← heapVector[ iSlotIndex ]
        if ( d(node) < d(leftChild) and d(node) < d(rightChild) )
                if ( d(leftChild) > d(rightChild) )
                        higher = left;
                else if ( d( rightChild ) > d( leftChild ) )
                        higher = right;
                end
        end if

        if ( not higher.isEmpty() )
                swap(higher, node)
                return
        end if

        if ( leftChild > node )
                swap(leftChild, node)
                return;
        end if

        if( rightChild > node )
```

```
            swap(leftChild, node)
            return;
      end if
end
```

**Tarefa 5:**

Para esta tarefa implementamos a classe VertexVectorAdjacencyList que possui o vetor (vectorVertex) de ponteiros para uma lista, cujo tamanho é o número de graus possíveis para um grafo (n - 1, onde n é o número de vértices). Guardamos também o índice deste vetor (lastHighestDegree) em que encontramos o vértice de maior grau, para que possamos iniciar deste ponto na próxima busca. Além disso, criamos um vetor auxiliar composto por ponteiros para os nós de lista (elementList) que nos permite o acesso direto aos vértices. Criamos também a classe List que é uma abstração de uma lista de inteiros (vertex e degree). O grau do vertice está no nó da lista, para que possamos reconhecer, a partir do nó da lista, em que lista estamos. Para o auxílio de alguns métodos da classe VertexVectorAdjacencyList, utilizamos os métodos erase() e remove() referentes à classe Lista, que também são especificados a seguir:

**Classe VertexVectorAdjacencyList:**

```
RemoveHighestDegreeVertex()
      vertex ← GetHighestDegreeVertex()
      for ( i ← vizinho de vertex )
            # ele pode ser vizinho mas não estar mais a lista de grau
            # caso ele já tenha sido processado anteriormente
            # vide observação tarefa 3

            if ( i ainda não foi removido do grafo )
                  DecrementDegree( i )
                  nEdges ← nEdges - 1
            end if
      end for

      RemoveFromVertexVector( vertex )
      return vertex
end

GetHighestDegreeVertex()
```

```
    for ( i ← lastHighestDegree .. 1 )
        if ( not vectorVertex[ i ].isEmpty )
            lastHighestDegree ← i
            return vectorVertex[ i ].firstElement
        end if
    end for
end


DecrementDegree( vertex )
    # acha o nó diretamente
    element ← elementList[ vertex ]
    degree ← element.degree

    # com o degree, podemos removê-lo da lista apropriada
    vectorVertex[ degree ].remove( element )

    # e inserir na lista imediatamente inferior
    node ← vectorVertex[ degree - 1 ].insert( vertex )

    # guarda o degree decrementado no novo nó
    node.degree ← degree - 1

    # atualiza o indice
    elementList[ vertex ] ← node
end

RemoveFromVertexVector( vertex )
    element ← elementList[ vertex ]

    vectorVertex[ element.degree ].remove(element)
    elementList[ vertex ] ← NULL
end
```

**Classe List:**

```
erase( vertex )
    for ( i ← 1 .. n - 1 )
        if ( i = vertex )
            detachFromList( i )
```

```
                nElements ← nElements - 1
                return
            end
        end for
end

remove( node )
        if ( node = NULL )  return

        detachFromList( node )
        nElements ← nElements - 1
end
```

**Análise de complexidade dos algoritmos**

Executamos no máximo n-1 chamadas ao método RemoveHighestDegreeVertex.

**Tarefa 3:**

Gastos de cada RemoveHighestDegreeVertex:
- O($n$) para encontrar o vértice de maior grau (GetHighestDegreeVertex) .

Globalmente:
- O( $m$ ) para decrementar o grau de vizinhos do vértice removido.
    - O(1) para decrementar o grau de um vértice (DecrementDegree).

Logo, RemoveHighestDegreeVertex gasta O($n$) e o algoritmo é O($n^2$ + $m$).

**Tarefa 4:**

Gastos para cada RemoveHighestDegreeVertex:
- O(1) para remover a raiz do heap(RemoveRoot).
- O($log\ n$) para balancear a heap(RemoveRoot).

Globalmente:
- O($m$) para decrementar o grau de todos os vizinhos.
    - O($log\ n$) para decrementar o grau de um vértice.

Logo, tempo de execução do algoritmo é = O($(n + m) . log\, n$)

**Tarefa 5:**

Gastos para cada RemoveHighestDegreeVertex:
- O(1) para encontrar o vértice de maior grau (GetHighestDegreeVertex).

Globalmente:
- O($m$) para decrementar o grau de todos os vizinhos dos vértices removidos.
    - O(1) para decrementar o grau de um vértice (DecrementDegree).
    - O(1) para tirar da lista do grau anterior
    - O(1) para inserir na lista do grau atual (decrementado)

Logo, tempo de execução do algoritmo é O($m$ + $n$).

**Observações:**

Comparando as implementações dos algoritmos, espera-se que os tempos de execução serão dispostos da seguinte forma: Tarefa 5 < Tarefa 3 < Tarefa 4.
A diferença entre os tempos de execução das tarefas 5 e 3 é dada pela diferentes constantes. Porém alguns fatores devem ser considerados:

Como o algoritmo guloso não garante sempre o mesmo resultado, as diferentes implementações retornam coberturas de diferentes tamanhos, dado que seus critérios de desempate não são os mesmos. Optamos por não garantir o mesmo critério de desempate, pois isso implicaria em um custo adicional para um dos algoritmos.
Dessa forma, deve-se levar em consideração que o algoritmo que retornou uma cobertura maior executou mais iterações e teve o tempo de execução maior.

Apesar do algoritmo 3 ter uma complexidade maior do que o algoritmo 5 em termos de n e m, a constante do primeiro é menor, tendo em vista que o acesso direto a um vetor é bem mais eficiente que a alocação e desalocação de nós de uma lista utilizando alocação de memória dinâmica.

Na prática, para grafos mais densos (como no caso de p = 0,2), o tempo de execução seguiu a seguinte ordem: 3 < 4 < 5. Nos grafos mais esparsos, tivemos o heap com melhor desempenho, seguido da lista de vértices de mesmo grau e por último o vetor de graus (4 < 3 < 5).

O tempo de carregamento do grafo, onde são de fato lidas as informações de vértices do arquivo e populadas as estruturas de dados específicas para cada algoritmo não variou muito de algoritmo para algoritmo (para os grafos analisados) de modo que podemos desprezar esta informação.

No Heap, apesar de utilizarmos log n (para o pior caso) como o fator de complexidade, como os graus são geralmente alterados de uma unidade, o conserto do heap acaba sendo bem rápido, o que nos permite aproximar para algo próximo de 1. O que explica a velocidade do algoritmo proposto.

Na estratégia de lista (tarefa 5), podemos perceber que o uso constante da primitiva malloc (em C++ new) diminui consideravalmente a performance do algoritmo, o que nos leva a concluir que seria necessário uma solução alternativa para esta alocação de memória, para diminuir as constantes e tirar proveito de sua eficiencia teórica.