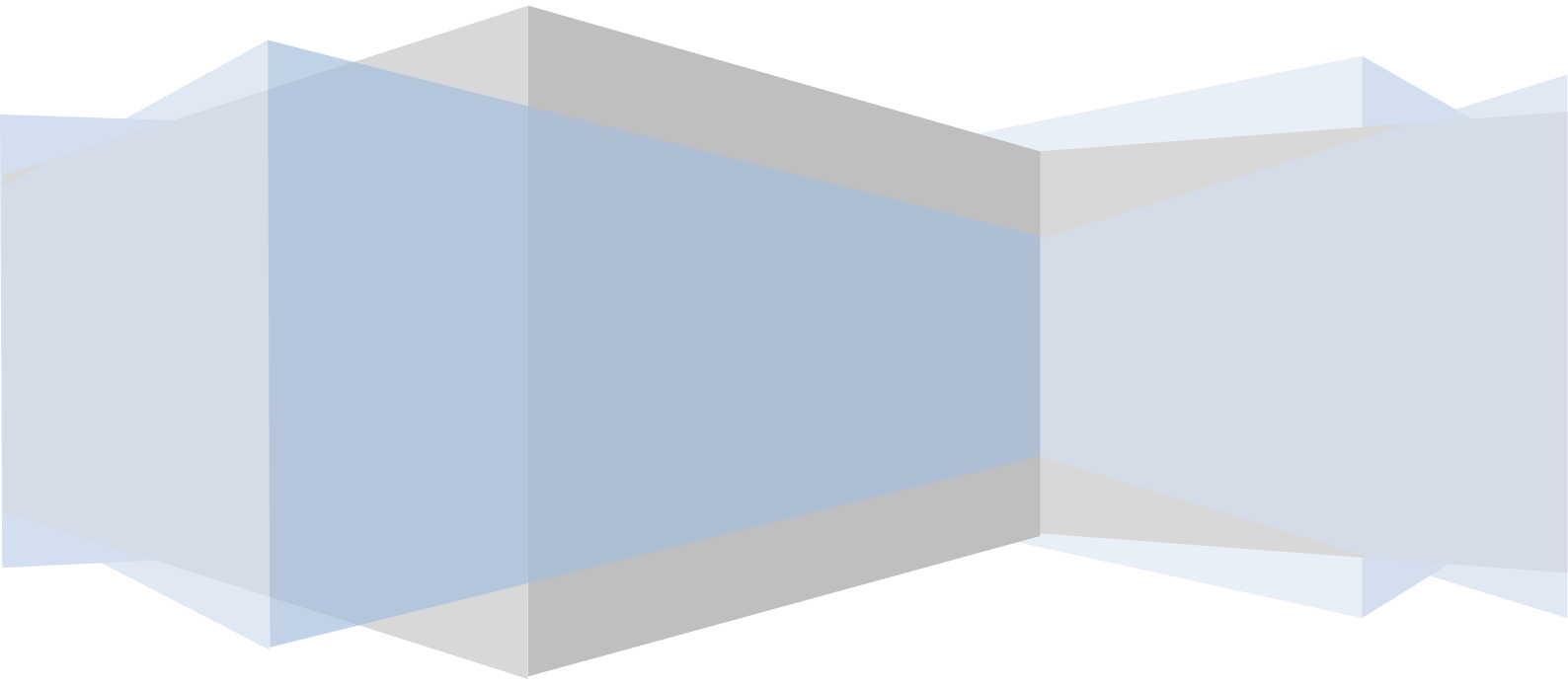
Faculdade de Ciências e Tecnologia da Universidade Nova de Lisboa

Análise e Desenho de Algoritmos

Resumo da Matéria v4.0

original de Gil Costa, adaptado, revisto e actualizado por Pedro Camelo.



Índice

Introdução

Este é um resumo da matéria da cadeira de Análise e Desenho de Algoritmos que aproveita de entre os slides da cadeira, material disponibilizado por colegas de curso.

Não pretendo que este seja um documento perfeito de início, deverá ser continuado por quem tenha interesse, poderei disponibilizar o ficheiro original a quem queira melhorar este resumo com a condição de que o mesmo seja divulgado e distribuído de forma pública e os autores base mantidos.

Espero que este resumo te seja útil, resta-me desejar um bom estudo e um bom exame ;)

"LEI é Rei!"

Agradecimentos:

Eduardo Costa – Revisão deste Documento;

Gil Costa – Versão Original do Resumo;

Gilherme "Guilhe" Delgado – Pelo Algoritmo Visual do algoritmo de *Prim* e detecção de Gralhas;

Hugo "Janado" Almeida – Pela SB, a baqueta e o valioso pergaminho de papel;

"Xocas" – Pela excelente grelha das Propriedades de Grafos e Algoritmos;

Pedro Camelo

13 de Junho de 2010

(Véspera de Exame de ADA)

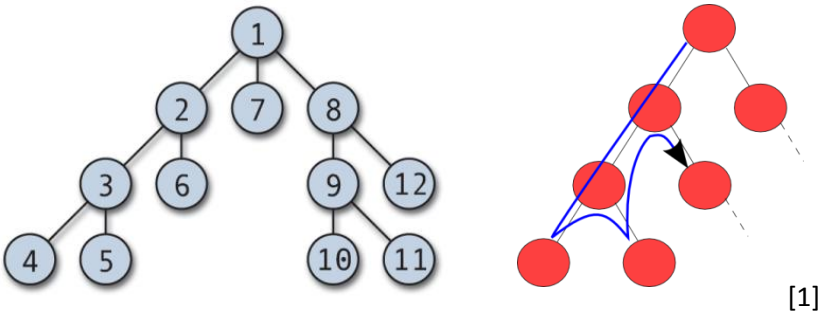
Propriedades de Grafos e Algoritmos

	Objectivo	Orientado?	c/ Ciclos?	Conexo?	Pesado?	Restrição
Percurso em Profundidade	Obter Percurso	Qualquer				
Percurso em Largura	Obter Percurso	Qualquer				
Ordenação Topológica	Ordena conjunto de arcos e obtêm um percurso	Sim	Não	Não		
Kruskal	Árvore de Cobertura Mínima (árvore com custo menor)	Não	Qualquer	Qualquer		
Prim	Árvore de Cobertura Mínima (árvore com custo menor)	Não	Qualquer	Qualquer		
Dijkstra	Caminho mais curto de um vértice a todos os outros	Sim	Qualquer	Qualquer	Sim	Pesos não Negativos
Bellman-Ford	Caminho mais curto de um vértice a todos os outros	Sim	Qualquer	Qualquer	Sim	
Edmonds-Karp	Fluxo máximo entre dois vértices	Sim			Sim	Pesos não Negativos

Percurso em Profundidade

Ideia Geral

O Percurso em profundidade é um algoritmo usado para realizar uma procura ou percurso numa árvore, estrutura de árvore ou grafo. Intuitivamente, o algoritmo começa pela raiz do grafo e explora tanto quanto possível cada um dos seus filhos antes de retroceder.



[1]

Ordem da procura através do percurso em profundidade.

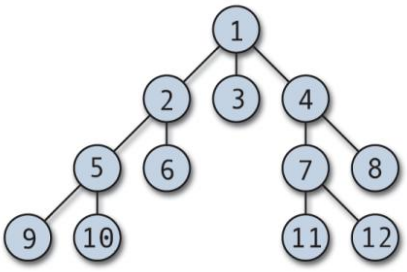
Implementação

```
void dfsTraversal( Graph graph ){
    boolean[] explored = new boolean[ graph.numVertices() ];
    for every Vertex v in graph.vertices()
        explored[v] = false;
        for every Vertex v in graph.vertices()
            if ( !explored[v] )
                dfsExplore(graph, explored, v);
}

// recursive implementation
void dfsExplore( Graph graph, boolean[] explored, Vertex root ){
    TREAT(root);
    explored[root] = true;
    for every Vertex v in graph.outAdjacentVertices(root)
        if ( !explored[v] )
            dfsExplore(graph, explored, v);
}

// iterative implementation
void dfsExplore( Graph graph, boolean[] explored, Vertex root ){
    Stack<Vertex> foundUnexplored = new StackInList<Vertex>();
    foundUnexplored.push(root);
    do {
        Vertex vertex = foundUnexplored.pop();
        if ( !explored[vertex] ){
            TREAT(vertex); explored[vertex] = true;
            for every Vertex w in graph.outAdjacentVertices(vertex)
                if ( !explored[w] )
                    foundUnexplored.push(w);
        }
    }
    while ( !foundUnexplored.isEmpty() );
}
```

Percurso em Largura

Ideia Geral

O Percurso em largura é um algoritmo de procura de árvore usado para realizar uma procura ou percurso numa árvore, estrutura de árvore ou grafo. Intuitivamente, o algoritmo começa pelo raiz do grafo. Para cada nó mais próximo explora os seus nós vizinhos inexplorados, até que ele encontre o objecto de procura ou chegue ao fim do grafo.



[2]

Ordem da procura através do percurso em largura.

Implementação

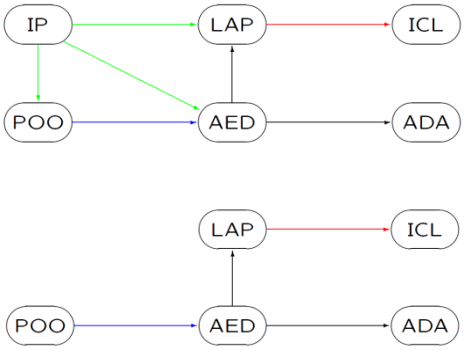
```
void dfsTraversal( Graph graph ){
      boolean[] explored = new boolean[ graph.numVertices() ];
      for every Vertex v in graph.vertices()
            explored[v] = false;
            for every Vertex v in graph.vertices()
                  if ( !explored[v] )
                        dfsExplore(graph, explored, v);
}
```

```
// iterative implementation
void bfsExplore( Graph graph, boolean[] found, Vertex root ){
      Queue<Vertex> waiting = new QueueInArray<Vertex>(
graph.numVertices() - 1 );
      waiting.enqueue(root);
      found[root] = true;
      do {
            Vertex vertex = waiting.dequeue(); TREAT(vertex);
            for every Vertex w in graph.outAdjacentVertices(vertex)
                  if ( !found[w] ){
                        waiting.enqueue(w);
                        found[w] = true;
                  }
      }
      while ( !waiting.isEmpty() );
}
```

Ordenação Topológica e Teste à Aciclicidade

Ideia Geral

Para cada vértice sem dependências, são eliminadas as dependências deste sobre outros vértices, e assim sucessivamente.



Número de Antecessores

LAP	2	1	1	**0**	—	—	—
ICL	1	1	1	1	**0**	—	—
AED	2	1	**0**	—	—	—	—
ADA	1	1	1	**0**	**0**	**0**	—
IP	**0**	—	—	—	—	—	—
POO	1	**0**	—	—	—	—	—
Ordenação:	IP	POO	AED	LAP	ICL	ADA	

Algoritmo

- Array de dependências (número de arcos incidentes em cada vértice);
- Fila de vértices a tratar, que (já) não têm dependências;
- Inicializar array e fila de acordo com os arcos incidentes em cada vértice;

- Enquanto fila não vazia
    a. Retirar um vértice;
    b. Tratar;
    c. Para cada vértice de destino dos arcos iniciados no vértice retirado;
        i. Decrementar as dependências da tabela;
        ii. Se for zero, adicionar à fila;

Implementação

```
void topologicalSort( Digraph graph ){
    Queue<Vertex> ready = new QueueInArray<Vertex>(graph.numVertices());
    int[] inCounter = new int[ graph.numVertices() ];
    for every Vertex v in graph.vertices(){
        inCounter[v] = graph.inDegree(v);
        if ( inCounter[v] == 0 )
        ready.enqueue(v);
    }
    while ( !ready.isEmpty() ){
        Vertex vertex = ready.dequeue();
        TREAT(vertex);
        for every Vertex w in graph.outAdjacentVertices(vertex){
            inCounter[w]--;
            if ( inCounter[w] == 0 )
                ready.enqueue(w);
        }
    }
}
```

Algoritmo de Kruskal

Ideia Geral

Partição para representar os componentes já conectados pela árvore. Para cada arco, por ordem crescente de peso, se os seus vértices pertencerem a conjuntos diferentes, aceitar arco e unir conjuntos (na prática, conectar duas componentes do grafo com o novo arco). Caso contrário, ignorar arco (vértices do arco já ligados por um caminho qualquer).

Algoritmo

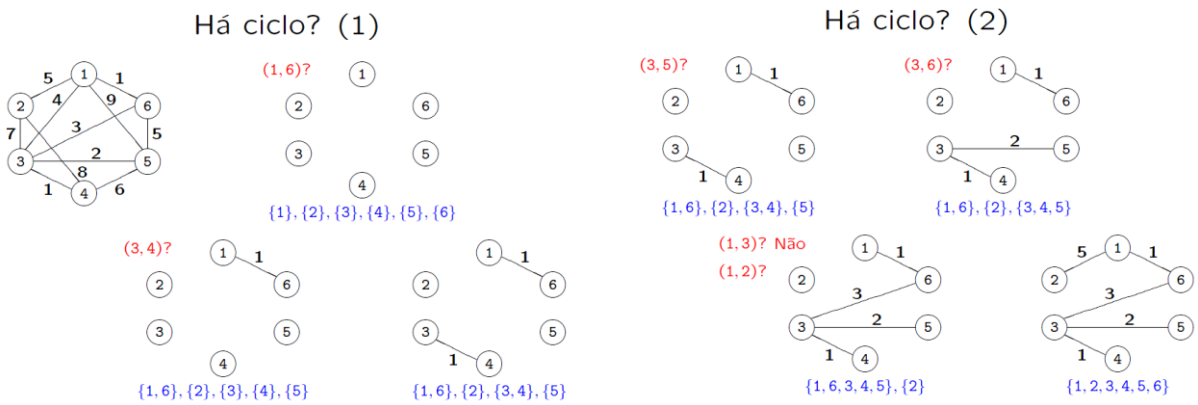
- Fila arcos ordenada crescentemente no peso dos arcos;
- Partição, criada com um conjunto ara cada vértice;
- Lista para guardar os arcos resultantes do algoritmo;

- Enquanto a lista ainda não estiver completa
  - Retirar próximo arco de custo mínimo da fila;
  - Encontrar os representantes dos dois vértices do arco na partição;
  - Se representantes diferentes
    - Unir os conjuntos desses representantes na partição;
    - Adicionar arco ao resultado;
  - caso contrário, ignora o arco;
- Devolver o arco;

Implementação

```
MinPriorityQueue<E,Edge<?,E>> buildEdgesPQ(UndiGraph<?,E> graph ){
    int size = graph.numEdges();
    Entry<E,Edge<?,E>>[] auxArray =
        (Entry<E,Edge<?,E>>[]) new Entry[size];
    int pos = 0;
    for every Edge<?,E> e in graph.edges()
        auxArray[pos++] = new EntryClass<E,Edge<?,E>>(e.label(), e);
        MinPriorityQueue<E,Edge<?,E>> priQueue =
            new MinHeap<E,Edge<?,E>>(size, auxArray, size);
    return priQueue;
}

Iterator<Edge<?,E>> mstKruskal( UndiGraph<?,E> graph ){
    MinPriorityQueue<E,Edge<?,E>> allEdges = buildEdgesPQ(graph);
    UnionFind vertPartition = new UnionFindInArray( graph.numVertices() );
    List<Edge<?,E>> mst = new DoublyLinkedList<Edge<?,E>>();
    int mstFinalSize = graph.numVertices() - 1;
    while ( mst.size() < mstFinalSize ){
        Edge<?,E> edge = allEdges.removeMin().getValue();
        Vertex<?>[] endPoints = edge.endVertices();
        int rep1 = vertPartition.find( endPoints[0] );
        int rep2 = vertPartition.find( endPoints[1] );
        if ( rep1 != rep2 ){
            mst.addLast(edge);
            vertPartition.union(rep1, rep2);
        }
    }
    return mst.iterator();
}
```

Algoritmo Visual



TAD Partição

Domínio = {0, 1, ..., n - 1}

```
// Cria a partição {{0}, {1}, . . . , {n - 1}}.
Partição cria( int n );
// Devolve o representante do conjunto ao qual e pertence.
Domínio representante( Domínio e );
// Substitui os conjuntos Ce e Cf , cujos representantes são e e f,
// respectivamente, pelo conjunto Ce U Cf .
// Pré-condição: e ≠ f (ou seja, Ce ≠ Cf ).
void união( Domínio e, Domínio f );
```

Interface Partição

```
public interface UnionFind{
    // Creates the partition {{0}, {1}, . . . , {domainSize - 1}}.
    // UnionFind( int domainSize );
    // Returns the representative of the set that contains
    // the specified element.
    int find( int element ) throws InvalidElementException;
    // Removes the two distinct sets S1 and S2 whose representatives
    // are the specified elements, and inserts the set S1 [ S2.
    // The representative of the new set S1 [ S2 can be any of
    // its members.
    void union( int representative1, int representative2 ) throws
        InvalidElementException, NotRepresentativeException,
        EqualSetsException;
}
```

Algoritmo de Prim

Ideia Geral

Este é um algoritmo que permite descobrir o caminho mínimo existente num grafo conexo e não orientado.

Algoritmo

Começar por um vértice qualquer; Pesquisar todos os arcos incidentes e, se o custo de ligar cada um dos vértices alcançados à árvore através desses arcos for inferior ao custo actual, actualizar o custo e o arco pelo qual o vértice é ligado à árvore. Se o vértice não estiver em fila de espera para ser tratado, introduzi-lo.

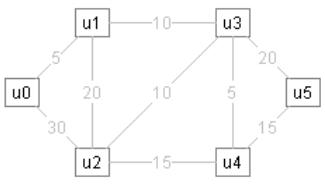
Caso contrário, decrementar o custo de espera (chave da fila); Em cada passo é sempre seleccionado o vértice ligado com custo de ligação mais baixo, o que garante que não há forma mais barata de ligar aquele vértice à rede, e o arco por onde se liga é adicionado ao resultado.

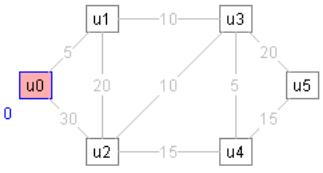
- Informação dos vértices já seleccionados (inicialmente nenhum seleccionado);
- Custo de ligar cada vértice à árvore (inicializado a +∞) e arco da ligação;
- Fila com prioridade adaptável de vértices já ligados, ordenada por custo;
- Lista para guardar os arcos resultantes do algoritmo;

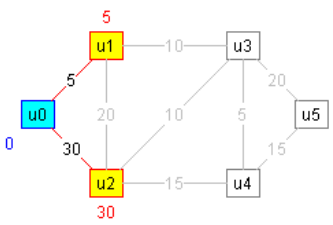
- O vértice inicial é um qualquer, tem custo zero e é introduzido na fila;
- Retirar enquanto a fila não está vazia (teste de condição no fim do ciclo);
    - Retirar um vértice da fila;
    - Seleccioná-lo;
    - Se não for a origem …
        - Adicionar ao resultado o arco por onde se chegou ao vértice;
    - Explorar vértice; *
- Devolver resultado;

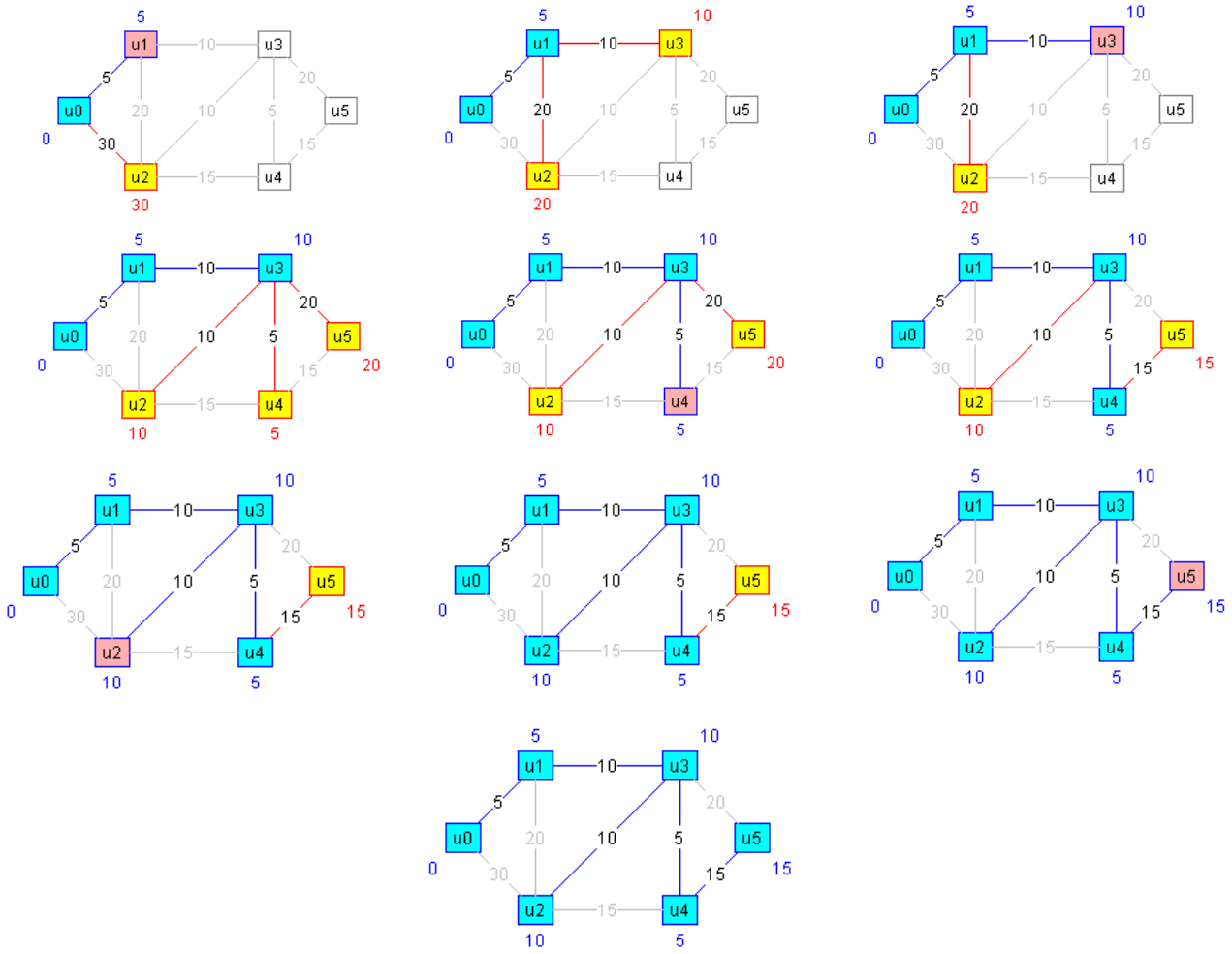
- * Explorar vértice;
    - Para cada arco incidente no vértice;
        - Obter o vértice na extremidade oposta;
        - Se o vértice não seleccionado e custo do arco inferior ao custo actual;
            - Actualiza-se o custo e o arco por onde se chega ao vértice;
            - Se vértice não está na fila (custo antigo = +∞);
                - Inserir vértice na fila;
            - c.c. devrementar a chave (custo) do vértice na fila com o novo valor;

Algoritmo Visual









Implementação

```
Iterator<Edge<?,E>> mstPrim( UndiGraph<?,E> graph ){
    boolean[] selected = new boolean[ graph.numVertices() ];
    E[] cost = new E[ graph.numVertices() ];
    Edge<?,E>[] via = new Edge<?,E>[ graph.numVertices() ];
    AdaptMinPriorityQueue<E,Vertex> connected =
        new AdaptMinHeap<E,Vertex>( graph.numVertices() );
    List<Edge<?,E>> mst = new DoublyLinkedList<Edge<?,E>>();

    for every Vertex v in graph.vertices(){
        selected[v] = false;
        cost[v] = +∞;
    }

    Vertex origin = graph.aVertex();
    cost[origin] = 0;
    connected.insert(cost[origin], origin);
    do {
        Vertex vertex = connected.removeMin().getValue();
        selected[vertex] = true;
        if ( vertex != origin )
            mst.addLast( via[vertex] );
        exploreVertex(graph, vertex, selected, cost, via, connected);
    }
    while ( !connected.isEmpty() );

    return mst.iterator();}
```

```
void exploreVertex( UndiGraph<?,E> graph, Vertex source,
     boolean[] selected, E[] cost, Edge<?,E>[] via,
     AdaptMinPriorityQueue<E,Vertex> connected ){

     for every Edge<?,E> e in graph.incidentEdges(source){
          Vertex vertex = e.oppositeVertex(source);
          if ( !selected[vertex] && e.label() < cost[vertex] ){
               boolean vertexIsInQueue = cost[vertex] < +∞;
               cost[vertex] = e.label();
               via[vertex] = e;
               if ( vertexIsInQueue )
                    connected.decreaseKey(vertex, cost[vertex]);
               else
               connected.insert(cost[vertex], vertex);
          }
     }
}
```

Algoritmo de Dijkstra

Ideia Geral

Em cada iteração seleccionar o vértice com custo de chegada mínimo (garantindo-se assim que não há forma mais barata de chegar a esse vértice), e calcular o custo de chegar a outros vértices por via deste. Se o custo for mais baixo, actualizar e via, e inserir vértice na fila ou actualizar prioridade caso o vértice já esteja na fila (ou seja, ligado à origem por algum caminho).

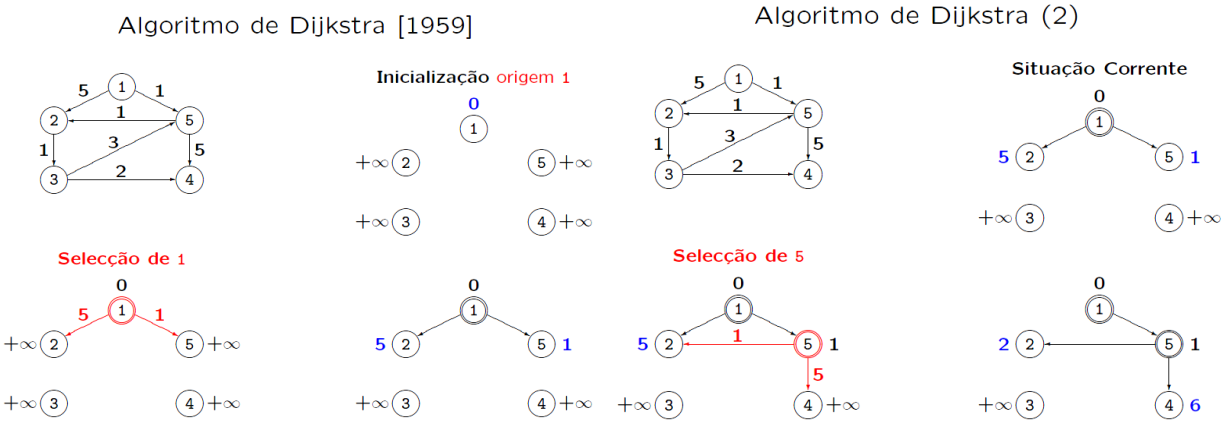
Algoritmo

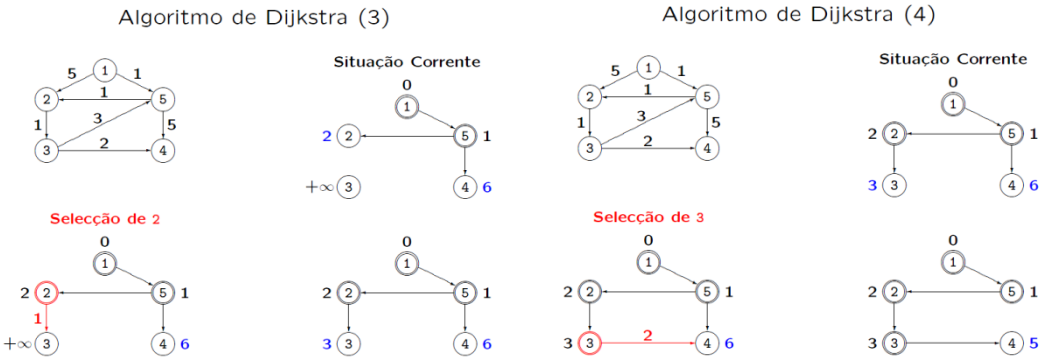
- Informação dos vértices já seleccionados (inicialmente nenhum seleccionado)
- Custo mínimo (actual) para ir da origem a cada vértice (+∞ s não há caminho)
- Via: penúltimo vértice no caminho da origem a cada vértice
- Fila com prioridades adaptável de vértices já ligados, ordenada por custo

- O custo de chegar à origem é zero, e chega-se por via dele próprio;
- Insere-se a origem na fila de vértices ligados;
- Repetir enquanto a fila não está vazia (teste de condição no fim):
    - Retirar um vértice da fila;
    - Seleccioná-lo;
    - Explorar vértice *;
- Devolver resultado (neste caso, par custos e via de todos os vértices)

- * Explorar vértice:
    - Para cada arco de extremidade inicial no vértice:
        - Obter o vértice na extremidade de chegada;
        - Se vértice não seleccionado
            - Calcula-se o custo de chegar a esse vértice por esse arco (custo de
            - chegar ao vértice corrente + custo de atravessar o arco);
            - Se o novo custo é inferior
                - Actualizar custo e vértice via
                - Se vértice não está na fila (custo antigo = +∞)
                    - Inserir vértice na fila;
                - c.c. decrementar chave (custo) do vértice na fila com o novo valor;

Algoritmo Visual





Implementação

```
Pair<E[], Vertex[]> dijkstra( Digraph<?,E> graph, Vertex origin ){
    boolean[] selected = new boolean[ graph.numVertices() ];
    E[] length = new E[ graph.numVertices() ];
    Vertex[] via = new Vertex[ graph.numVertices() ];
    AdaptMinPriorityQueue<E,Vertex> connected =
        new AdaptMinHeap<E,Vertex>( graph.numVertices() );

    for every Vertex v in graph.vertices(){
        selected[v] = false;
        length[v] = +∞;
    }

    length[origin] = 0;
    via[origin] = origin;
    connected.insert(length[origin], origin);
    do {
        Vertex vertex = connected.removeMin().getValue();
        selected[vertex] = true;
        exploreVertex(graph, vertex, selected, length, via, connected);
    }
    while ( !connected.isEmpty() );

    return new PairClass<E[], Vertex[]>(length, via);
}

void exploreVertex( Digraph<?,E> graph, Vertex source, boolean[] selected,
    E[] length, Vertex[] via, AdaptMinPriorityQueue<E,Vertex> connected ){

    for every Edge<?,E> e in graph.outIncidentEdges(source){
        Vertex vertex = e.endVertices()[1];
        if ( !selected[vertex] ){
            E newLength = length[source] + e.label();
            if ( newLength < length[vertex] ){
                boolean vertexIsInQueue = length[vertex] < +∞;
                length[vertex] = newLength;
                via[vertex] = source;
                if ( vertexIsInQueue )
                    connected.decreaseKey(vertex, length[vertex]);
                else
                    connected.insert(length[vertex], vertex);
            }
        }
    }}
```

Algoritmo de Bellman-Ford

Ideia Geral

Em cada passo, analisam-se todos os arcos do grafo. O conteúdo do vector nos passos intermédios depende da ordem pela qual os arcos são analisados.

Para cada arco iniciado num vértice já explorado (custo diferente de +∞) actualizar, caso seja inferior, o custo para o vértice de destino do arco.

Um caminho mais curto passa no máximo por #V vértices e #V-1 arcos. Portanto, actualização de custos só pode ocorrer no máximo #V-1 vezes. Se a actualização de custos executou #V-1 vezes com alterações, então o caminho é aceite sse na próxima iteração não ocorrerem alterações, caso contrário o grafo tem ciclos de peso negativo.

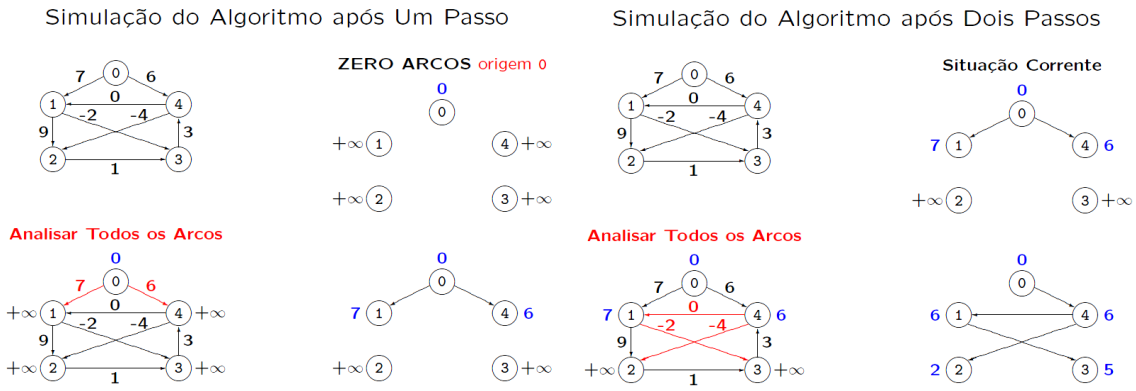
Algoritmo

- Custo mínimo (actual) para ir da origem a cada vértice (+∞ se não há caminho)
- Via: penúltimo vértice no caminho da origem a cada vértice

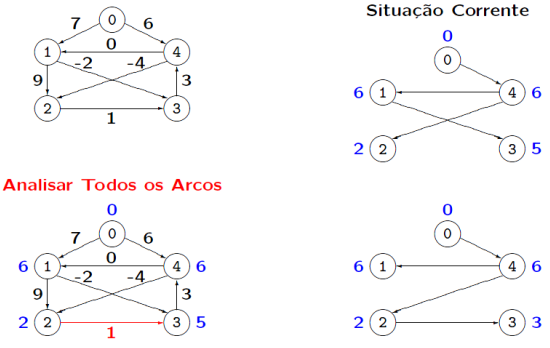
- O custo de chegar à origem é zero, e chega-se por via dele próprio.
- Iterar tantas vezes quanto o número de vértices menos um
  - Actualizar os custos;
  - Se não se verificaram alterações, sair do ciclo;
- Se ocorreram alterações na última iteração
  - Actualizar os custos *;
  - Se voltou a ocorrer alterações
    - enviar excepção: "Ciclo de peso negativo";
- Devolver resultado (neste caso, par custos e via de todos os vértices);

- * Actualizar os custos:
  - Assinalar alterações a falso.
  - Para cada arco (origem → destino) do grafo
    - Se o vértice origem for +∞, continuar ciclo (vértice não explorado);
    - Calcular custo para o vértice destino (custo do vértice origem + arco);
    - Se custo inferior ao custo actual do vértice destino
      - Actualizar custo e via com os novos valores;
      - Assinalar que houve alterações (alterações a verdadeiro);
  - Devolver valor de alterações.

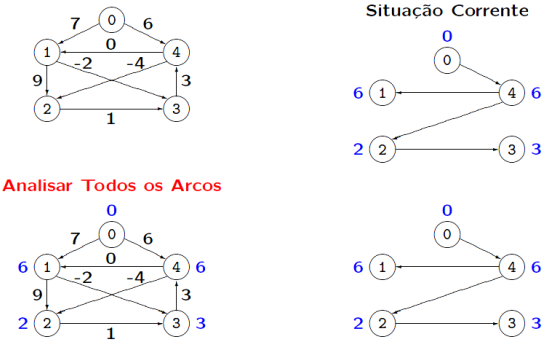
Algoritmo Visual



Simulação do Algoritmo após Três Passos



Simulação do Algoritmo após Quatro Passos



Implementação

```
Pair<E[], Vertex[]> bellmanFord( Digraph<?,E> graph, Vertex origin )
      throws NegativeWeightCycleException{

      E[] length = new E[ graph.numVertices() ];
      Vertex[] via = new Vertex[ graph.numVertices() ];

      for every Vertex v in graph.vertices()
            length[v] = +∞;

      length[origin] = 0;
      via[origin] = origin;
      boolean changes = false;
      for ( int i = 1; i < graph.numVertices(); i++ ){
            changes = updateLengths(graph, length, via);
            if ( !changes )
                  break;
      }

      // Negative-weight cycles detection.
      if ( changes && updateLengths(graph, length, via) )
            throw new NegativeWeightCycleException();
      else
            return new PairClass<E[], Vertex[]>(length, via);
}


boolean updateLengths( Digraph<?,E> graph, E[] length, Vertex[] via ){
      boolean changes = false;
      for every Edge<?,E> e in graph.edges(){
            Vertex[] endPoints = e.endVertices();
            if ( length[endPoints[0]] < +∞ ){
                  E newLength = length[endPoints[0]] + e.label();
                  if ( newLength < length[endPoints[1]] ){
                        length[endPoints[1]] = newLength;
                        via[endPoints[1]] = endPoints[0];
                        changes = true;
                  }
            }
      }

      return changes;
}
```

Algoritmo de Edmonds-Karp

Ideia Geral

Construir a rede residual, criando os arcos complementares com capacidade zero; Em cada passo, escolher um caminho por onde haja capacidade disponível (resíduo = capacidade total - fluxo corrente do arco), e guardar o mínimo dos resíduos obtidos (quanto é que efectivamente pode passar por esse caminho);

Actualizar ainda o fluxo total e os fluxos desse caminho através do resíduo (incrementar nos fluxos em sentido positivo, e decrementar nos fluxos em sentido negativo). Se o resíduo for zero, o fluxo é máximo. Múltiplas fontes/drenos: criar fonte/dreno que liga às outras com capacidade +∞.

Algoritmo

- Construir rede residual;
- - Informação do fluxo para cada arco (acessível pelos seus vértices, p.e. matriz);
- - Via: guarda o caminho efectuado por cada pesquisa, indicado para cada vértice,
- qual o vértice antecessor;
- - Valor do fluxo na rede;
- - Valor do incremento de fluxo após cada pesquisa
- Inicializar fluxos a zero;

- Enquanto o incremento resultante duma pesquisa de caminho for não nulo
    - Incrementar o fluxo da rede com o novo incremento;
    - Actualizar os fluxos dos arcos deste modo:
        - - Vértice auxiliar = dreno;
        - Enquanto vértice diferente de fonte:
            - Obter o antecessor do vértice através de via;
            - Incrementar o fluxo do arco {antecessor → vértice} de incremento;
            - Decrementar o fluxo do arco {vértice ← antecessor} de incremento;
            - Vértice auxiliar = antecessor;
- Devolver resultado. Neste caso, um par (fluxo da rede, fluxos dos arcos).

- Pesquisa de caminho:
    - - Fila de vértices em espera (tamanho máximo: #V-1);
    - - Informação de vértices encontrados inicializado a falso para todo o vértice;
    - - Informação do incremento de fluxo para cada vértice;

    - Colocar fonte na fila e marcá-la como encontrada;
    - Via da fonte é ela própria; Incremento na fonte é +∞;
    - Repetir enquanto fila não vazia (teste de condição no fim)
        - Retirar vértice da fila;
        - Para cada arco iniciado em vértice
            - Obter vértice de destino do arco;
            - valor residual = capacidade do arco – fluxo do arco (vértice → destino);

- se resíduo > 0 e destino ainda não "encontrado" (aceitar caminho)
    - via do destino = vértice;
    - incremento do destino = mínimo(incremento da origem, resíduo)
    - se destino é o dreno
        - devolver incremento do destino;
    - colocar destino na fila e marcá-lo "encontrado";

- Construir rede residual:
    - Para cada arco (origem → destino) do grafo
        - Inserir arco (destino → origem) com capacidade zero;

Implementação

```
Pair<E, E[][]> edmondsKarp( Digraph<?,E> graph, Vertex source, Vertex sink ){
    Digraph<?,E> network = buildNetwork(graph);
    int numVert = network.numVertices();
    E[][] flow = new E[numVert][numVert];

    for every Edge<?,E> e in network.edges(){
        Vertex[] endPoints = e.endVertices();
        flow[ endPoints[0] ][ endPoints[1] ] = 0;
    }

    Vertex[] via = new Vertex[numVert];
    E flowValue = 0;
    E increment;

    while ( ( increment = findPath(network, flow, source, sink, via) ) != 0 ){
        flowValue += increment;
        // Update flow.
        Vertex vertex = sink;
        while ( vertex != source ){
            Vertex origin = via[vertex];
            flow[origin][vertex] += increment;
            flow[vertex][origin] -= increment;
            vertex = origin;
        }
    }

    return new PairClass<E, E[][]>(flowValue, flow);
}
```

```
Digraph<?,E> buildNetwork( Digraph<?,E> graph ){
    // The network has every vertex and every edge in the graph.
    Digraph<?,E> network = graph.clone();
    // For every edge (v1, v2) in the graph (V,E), if (v2, v1) 62 E,
    // insert edge (v2, v1) with capacity zero in the network.
    for every Edge<?,E> e in graph.edges(){
        Vertex[] endPoints = e.endVertices();
        if ( !graph.edgeExists(endPoints[1], endPoints[0]) )
            network.insertEdge(endPoints[1], endPoints[0], 0);
```

```
    }
    return network;
}

E findPath( Digraph<?,E> network, E[][] flow, Vertex source, Vertex sink,
Vertex[] via ){
    int numVert = network.numVertices();
    Queue<Vertex> waiting =
        new QueueInArray<Vertex>(numVert - 1);
    boolean[] found = new boolean[numVert];

    for every Vertex v in network.vertices()
        found[v] = false;

    E[] pathIncr = new E[numVert];
    waiting.enqueue(source);
    found[source] = true;
    via[source] = source;
    pathIncr[source] = +∞;
    do {
        Vertex origin = waiting.dequeue();
        for every Edge<?,E> e in network.outIncidentEdges(origin){
            Vertex destin = e.endVertices()[1];
            E residue = e.label() - flow[origin][destin];
            if ( !found[destin] && residue > 0 ){
                via[destin] = origin;
                pathIncr[destin] = Math.min(pathIncr[origin], residue);
                if ( destin == sink )
                    return pathIncr[destin];
                else{
                    waiting.enqueue(destin);
                    found[destin] = true;
                }
            }
        }
    }
    while ( !waiting.isEmpty() );

    return 0;
}
```

Árvore Binária de Pesquisa com Promoção

Ideia Geral

Sempre que se acede a um nó k, k é promovido a raiz. No caso de pesquisas se k não existir, promove-se o pai da árvore vazia onde a pesquisa terminou, ou k c.c. Para inserções se k não existir, insere-se como folha e promove-se k. Promove-se k c.c. No caso de mínimos e máximos, promove-se o respectivo.

Remover k:

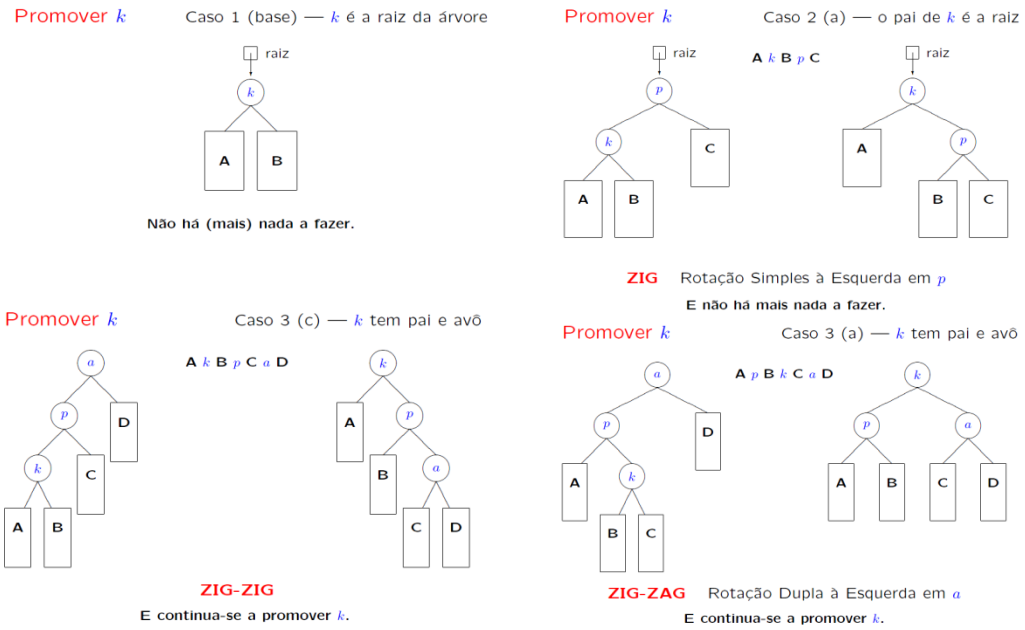
- Se k só tem uma sub-árvore não vazia, esta é a árvore resultante.
- Se as duas sub-árvores são não vazias:
  - Promove-se o mínimo da sub-árvore direita (fica como raiz da árvore final);
  - Liga-se a sub-árvore esquerda à raiz (mínimo da sub-árvore direita);
- Se ambas as sub-árvores de k são vazias, o resultado é árvore vazia.

Promover k:

- Se o pai de k é raiz: ZIG – rotação simples à direita ou à esquerda. (à esquerda quando k é filho esquerdo, ou à direita quando k é filho direito);
- Se o avô de k é raiz:
  - ZIGZAG quando:
  - pai é filho esquerdo do avô e k filho direito de pai, rotação dupla à esquerda
  - pai é filho direito do avô e k filho esquerdo de pai, rotação dupla à direita
  - ZIGZIG quando:
  - pai é filho direito de avô e k filho direito de pai, ou vice versa.

Na prática:

A ordem dos elementos da esquerda para a direita tem de ser a mesma antes e depois da promoção. Reconstruir a árvore: colocar k como raiz; O resto obtém-se ao respeitar a ordem dos os elementos.



Complexidades

Algoritmo	Implementação	Custo
Percurso em Profundidade Percurso em Largura Ordenação Topológica Teste à Aciclicidade	Matriz	$O((\#V)^2+(\#V) \times O(TREAT))$
	Vector de Listas	$O(\#V+\#A+(\#V) \times O(TREAT))$
Kruskal	União por Dimensão/Altura	$O(\#A \times \log(\#V))$
	Sem estratégia de União	$O(\#A \times \#V)$
Prim	Heap ou Binomial e Mapa	$O(\#A \times \log(\#V))$
	Fila de Fibonacci	$O(\#A + \#V \times \log(\#V))$
Dijkstra	Heap ou Binomial e Mapa	$O((\#V + \#A) \times \log(\#V))$
	Fila de Fibonacci	$O(\#A + \#V \times \log(\#V))$
Bellman-Ford	Vector de Listas	$O(\#V \times \#A)$
	Matriz de Adjacências	$O(\#V^3)$
Edmonds-Karp		$O(V \times (\#A)^2)$
BST com Promoção		Pior caso: O(n)
		Esperada e Amortizada O(log(n))

Complexidades da Fila com Prioridade Fundível

no pior caso e amortizadas (n entradas).

	Heap	Fila Binomial	Fila de Fibonacci
isEmpty	O(1)	O(1)	O(1)
size	O(1)	O(1)	O(1)
minEntry	O(1)	O(log n)	O(1)
Insert	O(log n)	O(log n)	O(1)
decreaseKey	O(log n)	O(log n)	O(1)
removeMin	O(log n)	O(log n)	O(log n)
remove	O(log n)	O(log n)	O(log n)
Merge (n entradas)	O(n + m)	O(log(n+m))	O(1)

Complexidade Amortizada

Define-se uma função potencial que atribui a cada Estrutura de Dados D, um número real $\phi(D)$. Normalmente corresponde ao número de elementos da ED.

Para $\phi F(F)$ ser uma função potencial válida tem que verificar duas condições:

- $\phi(D_o) = 0$ sendo D0 a estrutura de dados inicial; e
- $\phi(D_,) \geq 0$ para qualquer i, sendo D, a estrutura de dados depois da operação i.

Exemplo (apenas) para algumas operações de estruturas de dados clássicas:

Operação	Custo $C\iota$	Dif. de Potêncial $\Delta\phi = \Phi(T\iota) - \Phi(T\iota - 1)$	Custo Amortizado $C\hat{\iota} = C\iota + \Delta\phi$
isEmpty	O(1)	1 – 1 = 0	1 + 0 = O(1)
enqueue	O(1)	1 – (1 – 1) = 1	1 + 1 = 2 = O(1)
Clear	O(n)	0 – n = -n	n – n = 0 = O(1)

Problemas NP e Redução de Problemas

Problema NP

- Prova do sim: apresentar uma solução para uma instância do problema (normalmente há uma no enunciado);

- Provar que existe um algoritmo polinomial que resolve o problema:
  - Descrever o algoritmo;
  - Apresentar a sua complexidade, confirmando que é polinomial;

- Provar que a dimensão da solução é polinomial na dimensão do problema;

Problema NP-Difícil

- Provar que existe uma redução polinomial dum problema qualquer ao problema original:

  ϕ : probX ----------> prob original , ϕ polinomial

  Inst ----------> ϕ (Inst)

Problema NP-Completo

- Provar que o problema é NP e NP-Difícil.

Nota: utilizar os problemas dos acetatos para exemplos de demonstrações NP e bases para reduções.

Referências

Bibliográficas

Mamede, Margarida; Slides de Análise e Desenho de Algoritmos, DI FCT UNL 2010;

Costa, Gil; Resumos dos acetatos de ADA 07/08, 2008;

"Xocas", Apontamentos de ADA;

Wikipédia – A Enciclopédia Livre;

Imagens

Mamede, Margarida; Slides de Análise e Desenho de Agoritmos, DI FCT UNL 2010;

[1] – Depth first tree *in Wikipédia, Busca em Profundidade;*

[2] – Breadth first tree *in Wikipédia, Busca em Largura;*