Trabalho Prático Grafos

Gerado por Doxygen 1.10.0

Capítulo 1

Índice das estruturas de dados

1.1 Estruturas de dados

Lista das estruturas de dados com uma breve descrição:

Capítulo 2

Índice dos ficheiros

2.1 Lista de ficheiros

Lista de todos os ficheiros documentados com uma breve descrição:

Capítulo 3

Documentação da estruturas de dados

3.1 Referência à estrutura DFSContext

Variables that give context for the DFS function to work.

`#include <search.h>`

Campos de Dados

- const Graph * **graph**
- unsigned int * **pathVertices**
- unsigned int * **pathWeights**
- bool * **visited**
- unsigned int **pathIndex**
- struct PathNode ** **paths**
- unsigned int * **numPaths**
- unsigned int * **pathCapacity**

3.1.1 Descrição detalhada

Variables that give context for the DFS function to work.

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- Maximal-Graph-Sum/search.h

3.2 Referência à estrutura Edge

Structure of an edge in the graph which contains a destination vertex, weight and pointer to the next edge in the linked list.

`#include <graph-structure.h>`

Campos de Dados

- unsigned int **dest**
- unsigned int **weight**
- struct Edge * **next**

3.2.1 Descrição detalhada

Structure of an edge in the graph which contains a destination vertex, weight and pointer to the next edge in the linked list.

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- Maximal-Graph-Sum/graph-structure.h

3.3 Referência à estrutura Graph

Structure of a graph built with a hash table for vertices and linked lists for edges.

`#include <graph-structure.h>`

Campos de Dados

- unsigned int **numVertices**
- unsigned int **hashSize**
- Vertex ** **vertices**

3.3.1 Descrição detalhada

Structure of a graph built with a hash table for vertices and linked lists for edges.

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- Maximal-Graph-Sum/graph-structure.h

3.4 Referência à estrutura HeapNode

Represents a node in the heap used for Dijkstra's algorithm.

`#include <dijkstra-structure.h>`

Campos de Dados

- unsigned int **vertex**
- unsigned int **weight**

3.4.1 Descrição detalhada

Represents a node in the heap used for Dijkstra's algorithm.

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- Maximal-Graph-Sum/dijkstra-structure.h

3.5 Referência à estrutura MaxHeap

Max-heap data structure for finding the path with maximum weight.

`#include <dijkstra-structure.h>`

Campos de Dados

- HeapNode * **nodes**
- unsigned int **size**
- unsigned int **capacity**

3.5.1 Descrição detalhada

Max-heap data structure for finding the path with maximum weight.

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- Maximal-Graph-Sum/dijkstra-structure.h

3.6 Referência à estrutura MinHeap

Min-heap data structure for Dijkstra's algorithm.

`#include <dijkstra-structure.h>`

Campos de Dados

- HeapNode * **nodes**
- unsigned int **size**
- unsigned int **capacity**

3.6.1 Descrição detalhada

Min-heap data structure for Dijkstra's algorithm.

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- Maximal-Graph-Sum/dijkstra-structure.h

3.7 Referência à estrutura PathNode

A linked list structure which holds a path.

`#include <search.h>`

Campos de Dados

- unsigned int * **vertices**
- unsigned int * **weights**
- unsigned int **length**
- struct PathNode * **next**

3.7.1 Descrição detalhada

A linked list structure which holds a path.

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- Maximal-Graph-Sum/search.h

3.8 Referência à estrutura Vertex

Structure of a vertex of a graph which contains an identification number, a linked list of all edges and a next position to traverse to the next vertices.

`#include <graph-structure.h>`

Campos de Dados

- unsigned int **id**
- Edge * **edges**
- struct Vertex * **next**

3.8.1 Descrição detalhada

Structure of a vertex of a graph which contains an identification number, a linked list of all edges and a next position to traverse to the next vertices.

This structure contains a next field to traverse to other vertices which were hashed to the same position of the hash table. This solution is a simple yet effective way to handle collisions in the hash table.

A documentação para esta estrutura foi gerada a partir do seguinte ficheiro:

- Maximal-Graph-Sum/graph-structure.h

Capítulo 4

Documentação do ficheiro

4.1 Referência ao ficheiro Maximal-Graph-Sum/dijkstra-max.c

Function implementations of finding the path with maximum weight in a graph using Dijkstra's algorithm with a max-heap approach.

```
#include "dijkstra-max.h"
#include <limits.h>
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include "dijkstra-structure.h"
#include "vertices.h"
```

Funções

- MaxHeap ∗ CreateMaxHeap (unsigned int capacity)

  Creates a new MaxHeap with the specified capacity.
- void SwapHeapNodeMaxHeap (HeapNode ∗a, HeapNode ∗b)

  Swaps two HeapNode elements.
- void MaxHeapify (MaxHeap ∗heap, int idx)

  Maintains heap property for a given node.
- HeapNode ExtractMax (MaxHeap ∗heap)

  Extracts the maximum node (root) from the MaxHeap.
- void EnsureCapacityMaxHeap (MaxHeap ∗heap)

  Ensures the capacity of the MaxHeap.
- void InsertNodeMaxHeap (MaxHeap ∗heap, unsigned int vertex, unsigned int weight)

  Inserts a new node into the MaxHeap.
- void DijkstraMaxPath (const Graph ∗graph, unsigned int src, unsigned int dest, unsigned int ∗maxWeight, unsigned int ∗∗path, unsigned int ∗pathLength)

  Finds the path with maximum weight in a graph using a modified Dijkstra's algorithm.
- void PrintLongestPath (const unsigned int ∗path, unsigned int length, unsigned int maxWeight)

  Prints the path found by the DijkstraMaxPath function.

4.1.1 Descrição detalhada

Function implementations of finding the path with maximum weight in a graph using Dijkstra's algorithm with a max-heap approach.

Autor

Enrique George Rodrigues

Data

23.05.2024

Copyright

4.1.2 Documentação das funções

4.1.2.1 CreateMaxHeap()

```
MaxHeap * CreateMaxHeap (
            unsigned int capacity )
```

Creates a new MaxHeap with the specified capacity.

Parâmetros

capacity	The initial capacity of the MaxHeap.

Retorna

A pointer to the newly created MaxHeap, or NULL if memory allocation fails.

4.1.2.2 DijkstraMaxPath()

```
void DijkstraMaxPath (
            const Graph * graph,
            unsigned int src,
            unsigned int dest,
            unsigned int * maxWeight,
            unsigned int ** path,
            unsigned int * pathLength )
```

Finds the path with maximum weight in a graph using a modified Dijkstra's algorithm.

Parâmetros

graph	Pointer to the graph structure.

Parâmetros

src	Source vertex index.
dest	Destination vertex index.
maxWeight	Pointer to store the maximum path weight.
path	Pointer to store the path vertices with maximum weight.
pathLength	Pointer to store the length of the path.

4.1.2.3 EnsureCapacityMaxHeap()

```
void EnsureCapacityMaxHeap (
            MaxHeap * heap )
```

Ensures the capacity of the MaxHeap.

Parâmetros

heap	Pointer to the MaxHeap.

4.1.2.4 ExtractMax()

```
HeapNode ExtractMax (
            MaxHeap * heap )
```

Extracts the maximum node (root) from the MaxHeap.

Parâmetros

heap	Pointer to the MaxHeap.

Retorna

The maximum HeapNode extracted from the heap.

4.1.2.5 InsertNodeMaxHeap()

```
void InsertNodeMaxHeap (
            MaxHeap * heap,
            unsigned int vertex,
            unsigned int weight )
```

Inserts a new node into the MaxHeap.

Parâmetros

heap	Pointer to the MaxHeap.
vertex	Vertex index to insert.
weight	Weight associated with the vertex.

Nota

If the heap is full, it will be resized to accommodate more elements.

4.1.2.6 MaxHeapify()

```
void MaxHeapify (
            MaxHeap * heap,
            int idx )
```

Maintains heap property for a given node.

Parâmetros

heap	Pointer to the MaxHeap.
idx	Index of the node to start heapifying from.

4.1.2.7 PrintLongestPath()

```
void PrintLongestPath (
            const unsigned int * path,
            unsigned int length,
            unsigned int maxWeight )
```

Prints the path found by the DijkstraMaxPath function.

Parâmetros

path	Array of vertices representing the path.
length	Number of vertices in the path.
maxWeight	Maximum weight associated with the path.

4.1.2.8 SwapHeapNodeMaxHeap()

```
void SwapHeapNodeMaxHeap (
            HeapNode * a,
            HeapNode * b )
```

Swaps two HeapNode elements.

Parâmetros

a	Pointer to the first HeapNode.
b	Pointer to the second HeapNode.

4.2 Referência ao ficheiro Maximal-Graph-Sum/dijkstra-max.h

Function definitions for finding the path with maximum weight in a graph.

```
#include "dijkstra-structure.h"
#include "graph.h"
```

Funções

- MaxHeap * CreateMaxHeap (unsigned int capacity)
  Creates a new MaxHeap with the specified capacity.
- void SwapHeapNodeMaxHeap (HeapNode *a, HeapNode *b)
  Swaps two HeapNode elements.
- void MaxHeapify (MaxHeap *heap, int idx)
  Maintains heap property for a given node.
- HeapNode ExtractMax (MaxHeap *heap)
  Extracts the maximum node (root) from the MaxHeap.
- void EnsureCapacityMaxHeap (MaxHeap *heap)
  Ensures the capacity of the MaxHeap.
- void InsertNodeMaxHeap (MaxHeap *heap, unsigned int vertex, unsigned int weight)
  Inserts a new node into the MaxHeap.
- void DijkstraMaxPath (const Graph *graph, unsigned int src, unsigned int dest, unsigned int *maxWeight, unsigned int **path, unsigned int *pathLength)
  Finds the path with maximum weight in a graph using a modified Dijkstra's algorithm.
- void PrintLongestPath (const unsigned int *path, unsigned int length, unsigned int maxWeight)
  Prints the path found by the DijkstraMaxPath function.

4.2.1 Descrição detalhada

Function definitions for finding the path with maximum weight in a graph.

Autor

Enrique George Rodrigues

Data

23.05.2024

Copyright

4.2.2 Documentação das funções

4.2.2.1 CreateMaxHeap()

```
MaxHeap * CreateMaxHeap (
            unsigned int capacity )
```

Creates a new MaxHeap with the specified capacity.

Parâmetros

capacity	The initial capacity of the MaxHeap.

Retorna

A pointer to the newly created MaxHeap, or NULL if memory allocation fails.

4.2.2.2 DijkstraMaxPath()

```
void DijkstraMaxPath (
            const Graph * graph,
            unsigned int src,
            unsigned int dest,
            unsigned int * maxWeight,
            unsigned int ** path,
            unsigned int * pathLength )
```

Finds the path with maximum weight in a graph using a modified Dijkstra's algorithm.

Parâmetros

graph	Pointer to the graph structure.
src	Source vertex index.
dest	Destination vertex index.
maxWeight	Pointer to store the maximum path weight.
path	Pointer to store the path vertices with maximum weight.
pathLength	Pointer to store the length of the path.

4.2.2.3 EnsureCapacityMaxHeap()

```
void EnsureCapacityMaxHeap (
            MaxHeap * heap )
```

Ensures the capacity of the MaxHeap.

Parâmetros

heap	Pointer to the MaxHeap.

4.2.2.4 ExtractMax()

```
HeapNode ExtractMax (
            MaxHeap * heap )
```

Extracts the maximum node (root) from the MaxHeap.

Parâmetros

heap	Pointer to the MaxHeap.

Retorna

The maximum HeapNode extracted from the heap.

4.2.2.5 InsertNodeMaxHeap()

```
void InsertNodeMaxHeap (
            MaxHeap * heap,
            unsigned int vertex,
            unsigned int weight )
```

Inserts a new node into the MaxHeap.

Parâmetros

heap	Pointer to the MaxHeap.
vertex	Vertex index to insert.
weight	Weight associated with the vertex.

Nota

If the heap is full, it will be resized to accommodate more elements.

4.2.2.6 MaxHeapify()

```
void MaxHeapify (
            MaxHeap * heap,
            int idx )
```

Maintains heap property for a given node.

Parâmetros

heap	Pointer to the MaxHeap.
idx	Index of the node to start heapifying from.

4.2.2.7 PrintLongestPath()

```
void PrintLongestPath (
            const unsigned int * path,
            unsigned int length,
            unsigned int maxWeight )
```

Prints the path found by the DijkstraMaxPath function.

Parâmetros

path	Array of vertices representing the path.
length	Number of vertices in the path.
maxWeight	Maximum weight associated with the path.

4.2.2.8 SwapHeapNodeMaxHeap()

```
void SwapHeapNodeMaxHeap (
              HeapNode * a,
              HeapNode * b )
```

Swaps two HeapNode elements.

Parâmetros

a	Pointer to the first HeapNode.
b	Pointer to the second HeapNode.

4.3 dijkstra-max.h

Ir para a documentação deste ficheiro.

```
00001
00011 #ifndef DIJKSTRA_MAX_H
00012 #define DIJKSTRA_MAX_H
00013
00014 #include "dijkstra-structure.h"
00015 #include "graph.h"
00016
00023 MaxHeap* CreateMaxHeap(unsigned int capacity);
00024
00031 void SwapHeapNodeMaxHeap(HeapNode* a, HeapNode* b);
00032
00039 void MaxHeapify(MaxHeap* heap, int idx);
00040
00048 HeapNode ExtractMax(MaxHeap* heap);
00049
00055 void EnsureCapacityMaxHeap(MaxHeap* heap);
00056
00066 void InsertNodeMaxHeap(MaxHeap* heap, unsigned int vertex, unsigned int weight);
00067
00078 void DijkstraMaxPath(const Graph* graph, unsigned int src, unsigned int dest,
00079   unsigned int* maxWeight, unsigned int** path,
00080   unsigned int* pathLength);
00081
00089 void PrintLongestPath(const unsigned int* path, unsigned int length,
00090   unsigned int maxWeight);
00091
00092 #endif  // !DIJKSTRA_MAX_H
```

4.4 Referência ao ficheiro Maximal-Graph-Sum/dijkstra-min.c

Function implementations of Dijkstra algorithm to find shortest path.

```
#include "dijkstra-min.h"
#include <limits.h>
```

```
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include "dijkstra-structure.h"
#include "vertices.h"
```

Funções

- MinHeap ∗ CreateMinHeap (unsigned int capacity)

  Creates a new MinHeap with the specified capacity.
- void SwapHeapNode (HeapNode ∗a, HeapNode ∗b)

  Swaps two HeapNode elements.
- void MinHeapify (MinHeap ∗heap, int idx)

  Maintains heap property for a given node.
- HeapNode ExtractMin (MinHeap ∗heap)

  Extracts the minimum node (root) from the MinHeap.
- void EnsureCapacity (MinHeap ∗heap)

  Ensures the capacity of the MinHeap.
- void InsertNode (MinHeap ∗heap, unsigned int vertex, unsigned int weight)

  Inserts a new node into the MinHeap.
- void DijkstraMinPath (const Graph ∗graph, unsigned int src, unsigned int dest, unsigned int ∗minSum, unsigned int ∗∗path, unsigned int ∗pathLength)

  Computes the shortest path in a graph using Dijkstra's algorithm.
- void PrintShortestPath (const unsigned int ∗path, unsigned int length, unsigned int minSum)

  Prints the shortest path found by Dijkstra's algorithm.

4.4.1 Descrição detalhada

Function implementations of Dijkstra algorithm to find shortest path.

Autor

Enrique George Rodrigues

Data

23.05.2024

Copyright

4.4.2 Documentação das funções

4.4.2.1 CreateMinHeap()

```
MinHeap * CreateMinHeap (
            unsigned int capacity )
```

Creates a new MinHeap with the specified capacity.

Parâmetros

capacity	The initial capacity of the MinHeap.

Retorna

A pointer to the newly created MinHeap, or NULL if memory allocation fails.

4.4.2.2 DijkstraMinPath()

```
void DijkstraMinPath (
            const Graph * graph,
            unsigned int src,
            unsigned int dest,
            unsigned int * minSum,
            unsigned int ** path,
            unsigned int * pathLength )
```

Computes the shortest path in a graph using Dijkstra's algorithm.

Parâmetros

graph	Pointer to the graph structure.
src	Source vertex index.
dest	Destination vertex index.
minSum	Pointer to store the minimum path sum.
path	Pointer to store the shortest path vertices.
pathLength	Pointer to store the length of the shortest path.

4.4.2.3 EnsureCapacity()

```
void EnsureCapacity (
            MinHeap * heap )
```

Ensures the capacity of the MinHeap.

Parâmetros

heap	Pointer to the MinHeap.

4.4.2.4 ExtractMin()

```
HeapNode ExtractMin (
            MinHeap * heap )
```

Extracts the minimum node (root) from the MinHeap.

Parâmetros

heap	Pointer to the MinHeap.

Retorna

The minimum HeapNode extracted from the heap.

4.4.2.5 InsertNode()

```
void InsertNode (
            MinHeap * heap,
            unsigned int vertex,
            unsigned int weight )
```

Inserts a new node into the MinHeap.

Parâmetros

heap	Pointer to the MinHeap.
vertex	Vertex index to insert.
weight	Weight associated with the vertex.

Nota

If the heap is full, it will be resized to accommodate more elements.

4.4.2.6 MinHeapify()

```
void MinHeapify (
            MinHeap * heap,
            int idx )
```

Maintains heap property for a given node.

Parâmetros

heap	Pointer to the MinHeap.
idx	Index of the node to start heapifying from.

4.4.2.7 PrintShortestPath()

```
void PrintShortestPath (
            const unsigned int * path,
            unsigned int length,
            unsigned int minSum )
```

Prints the shortest path found by Dijkstra's algorithm.

Parâmetros

path	Array of vertices representing the shortest path.
length	Number of vertices in the shortest path.
minSum	Minimum path sum associated with the shortest path.

4.4.2.8 SwapHeapNode()

```
void SwapHeapNode (
            HeapNode * a,
            HeapNode * b )
```

Swaps two HeapNode elements.

Parâmetros

a	Pointer to the first HeapNode.
b	Pointer to the second HeapNode.

4.5 Referência ao ficheiro Maximal-Graph-Sum/dijkstra-min.h

Function definitions for Dijkstra algorithm to find the shortest path.

```
#include "dijkstra-structure.h"
#include "graph.h"
```

Funções

- MinHeap ∗ CreateMinHeap (unsigned int capacity)

  Creates a new MinHeap with the specified capacity.
- void SwapHeapNode (HeapNode ∗a, HeapNode ∗b)

  Swaps two HeapNode elements.
- void MinHeapify (MinHeap ∗heap, int idx)

  Maintains heap property for a given node.
- HeapNode ExtractMin (MinHeap ∗heap)

  Extracts the minimum node (root) from the MinHeap.
- void EnsureCapacity (MinHeap ∗heap)

  Ensures the capacity of the MinHeap.
- void InsertNode (MinHeap ∗heap, unsigned int vertex, unsigned int weight)

  Inserts a new node into the MinHeap.
- void DijkstraMinPath (const Graph ∗graph, unsigned int src, unsigned int dest, unsigned int ∗minSum, unsigned int ∗∗path, unsigned int ∗pathLength)

  Computes the shortest path in a graph using Dijkstra's algorithm.
- void PrintShortestPath (const unsigned int ∗path, unsigned int length, unsigned int minSum)

  Prints the shortest path found by Dijkstra's algorithm.

4.5.1 Descrição detalhada

Function definitions for Dijkstra algorithm to find the shortest path.

Autor

Enrique George Rodrigues

Data

23.05.2024

Copyright

4.5.2 Documentação das funções

4.5.2.1 CreateMinHeap()

```
MinHeap * CreateMinHeap (
            unsigned int capacity )
```

Creates a new MinHeap with the specified capacity.

Parâmetros

capacity	The initial capacity of the MinHeap.

Retorna

A pointer to the newly created MinHeap, or NULL if memory allocation fails.

4.5.2.2 DijkstraMinPath()

```
void DijkstraMinPath (
            const Graph * graph,
            unsigned int src,
            unsigned int dest,
            unsigned int * minSum,
            unsigned int ** path,
            unsigned int * pathLength )
```

Computes the shortest path in a graph using Dijkstra's algorithm.

Parâmetros

graph	Pointer to the graph structure.
src	Source vertex index.

Parâmetros

dest	Destination vertex index.
minSum	Pointer to store the minimum path sum.
path	Pointer to store the shortest path vertices.
pathLength	Pointer to store the length of the shortest path.

4.5.2.3 EnsureCapacity()

```
void EnsureCapacity (
            MinHeap * heap )
```

Ensures the capacity of the MinHeap.

Parâmetros

heap	Pointer to the MinHeap.

4.5.2.4 ExtractMin()

```
HeapNode ExtractMin (
            MinHeap * heap )
```

Extracts the minimum node (root) from the MinHeap.

Parâmetros

heap	Pointer to the MinHeap.

Retorna

The minimum HeapNode extracted from the heap.

4.5.2.5 InsertNode()

```
void InsertNode (
            MinHeap * heap,
            unsigned int vertex,
            unsigned int weight )
```

Inserts a new node into the MinHeap.

Parâmetros

heap	Pointer to the MinHeap.
vertex	Vertex index to insert.
weight	Weight associated with the vertex.

Nota

If the heap is full, it will be resized to accommodate more elements.

4.5.2.6 MinHeapify()

```
void MinHeapify (
            MinHeap * heap,
            int idx )
```

Maintains heap property for a given node.

Parâmetros

heap	Pointer to the MinHeap.
idx	Index of the node to start heapifying from.

4.5.2.7 PrintShortestPath()

```
void PrintShortestPath (
            const unsigned int * path,
            unsigned int length,
            unsigned int minSum )
```

Prints the shortest path found by Dijkstra's algorithm.

Parâmetros

path	Array of vertices representing the shortest path.
length	Number of vertices in the shortest path.
minSum	Minimum path sum associated with the shortest path.

4.5.2.8 SwapHeapNode()

```
void SwapHeapNode (
            HeapNode * a,
            HeapNode * b )
```

Swaps two HeapNode elements.

Parâmetros

a	Pointer to the first HeapNode.
b	Pointer to the second HeapNode.

4.6 dijkstra-min.h

Ir para a documentação deste ficheiro.

```
00001
00011 #ifndef DIJKSTRA_MIN_H
00012 #define DIJKSTRA_MIN_H
00013
00014 #include "dijkstra-structure.h"
00015 #include "graph.h"
00016
00023 MinHeap* CreateMinHeap(unsigned int capacity);
00024
00031 void SwapHeapNode(HeapNode* a, HeapNode* b);
00032
00039 void MinHeapify(MinHeap* heap, int idx);
00040
00048 HeapNode ExtractMin(MinHeap* heap);
00049
00055 void EnsureCapacity(MinHeap* heap);
00056
00066 void InsertNode(MinHeap* heap, unsigned int vertex, unsigned int weight);
00067
00077 void DijkstraMinPath(const Graph* graph, unsigned int src, unsigned int dest,
00078   unsigned int* minSum, unsigned int** path,
00079   unsigned int* pathLength);
00080
00088 void PrintShortestPath(const unsigned int* path, unsigned int length,
00089   unsigned int minSum);
00090
00091 #endif  // !DIJKSTRA_MIN_H
```

4.7 Referência ao ficheiro Maximal-Graph-Sum/dijkstra-structure.h

Structure definitions for Dijkstra algorithm.

Estruturas de Dados

- struct HeapNode

  Represents a node in the heap used for Dijkstra's algorithm.
- struct MinHeap

  Min-heap data structure for Dijkstra's algorithm.
- struct MaxHeap

  Max-heap data structure for finding the path with maximum weight.

Definições de tipos

- typedef struct HeapNode **HeapNode**
- typedef struct MinHeap **MinHeap**
- typedef struct MaxHeap **MaxHeap**

4.7.1 Descrição detalhada

Structure definitions for Dijkstra algorithm.

Autor

Enrique George Rodrigues

Data

23.05.2024

Copyright

4.8 dijkstra-structure.h

Ir para a documentação deste ficheiro.

```
00001
00010 #ifndef DIJKSTRA_STRUCTURE_H
00011 #define DIJKSTRA_STRUCTURE_H
00012
00017 typedef struct HeapNode {
00018   unsigned int vertex;  // vertex index
00019   unsigned int weight;  // vertex weight
00020 } HeapNode;
00021
00026 typedef struct MinHeap {
00027   HeapNode* nodes;        // Array of HeapNode elements
00028   unsigned int size;      // Number of elements in the heap
00029   unsigned int capacity;  // Maximum capacity of heap
00030 } MinHeap;
00031
00036 typedef struct MaxHeap {
00037   HeapNode* nodes;        // Array of heap nodes
00038   unsigned int size;      // Current number of elements in heap
00039   unsigned int capacity;  // Capacity of heap
00040 } MaxHeap;
00041
00042 #endif  // !DIJKSTRA_STRUCTURE_H
```

4.9 Referência ao ficheiro Maximal-Graph-Sum/edges.c

Function implementations for edge creation, deletion and management.

```
#include "edges.h"
#include <stdbool.h>
#include <stdlib.h>
#include "graph.h"
#include "vertices.h"
```

Funções

- Edge * CreateEdge (unsigned int dest, unsigned int weight)

  Creates a new edge with the specified destination and weight.
- bool AddEdgeToVertex (Vertex *vertex, Edge *edge)

  Adds an edge to a vertex.
- bool CreateAddEdge (Vertex *vertex, unsigned int dest, unsigned int weight)

  Creates an edge and adds it to a vertex.
- bool EdgeExists (Vertex *vertex, unsigned int dest)

  Checks if an edge exists.
- bool EdgeExistsBetweenVertices (const Graph *graph, unsigned int src, unsigned int dest)

  Checks if an edge exists between two given vertex ID's.
- int RemoveEdge (Vertex *vertex, unsigned int dest)

  Removes a specific edge from a vertex.
- int RemoveOutgoingEdges (Vertex *vertex)

  Removes all outgoing edges from a vertex.
- int RemoveEdgesPointingTo (Vertex *vertex, unsigned int targetVertexId)

  Removes edges pointing to a specific vertex from a vertex's edge list.
- int RemoveIncomingEdges (const Graph *graph, unsigned int vertexId)

  Removes all incoming edges to a specified vertex in the graph.

4.9.1 Descrição detalhada

Function implementations for edge creation, deletion and management.

Autor

Enrique Rodrigues

Data

22.05.2024

Copyright

4.9.2 Documentação das funções

4.9.2.1 AddEdgeToVertex()

```
bool AddEdgeToVertex (
            Vertex * vertex,
            Edge * edge )
```

Adds an edge to a vertex.

Parâmetros

vertex	- The vertex to which we add the edge.
edge	- The edge to be added.

Valores retornados

-	True in the case of success.
-	False if vertex or edge are NULL.

4.9.2.2 CreateAddEdge()

```
bool CreateAddEdge (
            Vertex * vertex,
            unsigned int dest,
            unsigned int weight )
```

Creates an edge and adds it to a vertex.

Parâmetros

vertex	- The vertex which will have the new edge.
dest	- The destination of the edge.
weight	- The weight of the edge.

Valores retornados

-	True if the edge was succesfully created and added.
-	False if edge already exists or in the event of an error.

4.9.2.3 CreateEdge()

```
Edge * CreateEdge (
            unsigned int dest,
            unsigned int weight )
```

Creates a new edge with the specified destination and weight.

Parâmetros

dest	- The destination of this edge.
weight	- The weight of this edge.

Valores retornados

-	A pointer to the newly created edge.
-	NULL if memory allocation fails.

4.9.2.4 EdgeExists()

```
bool EdgeExists (
            Vertex * vertex,
            unsigned int dest )
```

Checks if an edge exists.

Parâmetros

vertex	- The vertex where the edge starts.
dest	- The destination of the edge.

Valores retornados

-	True if the edge exists.
-	False if the edge does not exist.

4.9.2.5 EdgeExistsBetweenVertices()

```
bool EdgeExistsBetweenVertices (
            const Graph * graph,
```

```
            unsigned int src,
            unsigned int dest )
```

Checks if an edge exists between two given vertex ID's.

Parâmetros

graph	- The graph which contains the vertices and edges.
src	- The identifier of the source vertex.
dest	- The identifier of the destination vertex.

Valores retornados

-	True if the edge exists.
-	False if the edge does not exist.

4.9.2.6 RemoveEdge()

```
int RemoveEdge (
            Vertex * vertex,
            unsigned int dest )
```

Removes a specific edge from a vertex.

Parâmetros

vertex	- The vertex which contains the edge.
dest	- The destination of the edge.

Valores retornados

-	SUCCESS_REMOVING_EDGE if the edge was removed.
-	INVALID_VERTEX if the vertex is invalid.
-	EDGE_DOES_NOT_EXIST if the edge does not exist.
-	UNDEFINED_ERROR if edge was not found.

4.9.2.7 RemoveEdgesPointingTo()

```
int RemoveEdgesPointingTo (
            Vertex * vertex,
            unsigned int targetVertexId )
```

Removes edges pointing to a specific vertex from a vertex's edge list.

Parâmetros

vertex	- The vertex from which to remove edges.
target↩ VertexId	- The ID of the vertex to which the edges point.

Valores retornados

-	SUCCESS_REMOVING_EDGES if all edges were removed.
-	ERROR_REMOVING_EDGE if there was an error removing an edge.

4.9.2.8 RemoveIncomingEdges()

```
int RemoveIncomingEdges (
            const Graph * graph,
            unsigned int vertexId )
```

Removes all incoming edges to a specified vertex in the graph.

Parâmetros

graph	- The graph from which to remove incoming edges.
vertex↩ Id	- The ID of the vertex for which to remove incoming edges.

Valores retornados

-	SUCCESS_REMOVING_INCOMING_EDGES if all edges were removed.
-	INVALID_GRAPH if the graph is NULL.
-	ERROR_REMOVING_EDGE if there was an error removing an edge.

4.9.2.9 RemoveOutgoingEdges()

```
int RemoveOutgoingEdges (
            Vertex * vertex )
```

Removes all outgoing edges from a vertex.

Parâmetros

vertex	- The vertex from which to remove all outgoing edges.

Valores retornados

-	SUCCESS_REMOVING_OUTGOING_EDGES if all edges were removed.
-	VERTEX_EDGES_NULL if the vertex or its edges are NULL.
-	ERROR_REMOVING_EDGE if there was an error removing an edge.

4.10 Referência ao ficheiro Maximal-Graph-Sum/edges.h

Function definitions for edge creation, deletion and management.

```
#include <stdbool.h>
#include "graph.h"
```

Funções

- Edge ∗ CreateEdge (unsigned int dest, unsigned int weight)
  Creates a new edge with the specified destination and weight.
- bool AddEdgeToVertex (Vertex ∗vertex, Edge ∗edge)
  Adds an edge to a vertex.
- bool CreateAddEdge (Vertex ∗vertex, unsigned int dest, unsigned int weight)
  Creates an edge and adds it to a vertex.
- bool EdgeExists (Vertex ∗vertex, unsigned int dest)
  Checks if an edge exists.
- bool EdgeExistsBetweenVertices (const Graph ∗graph, unsigned int src, unsigned int dest)
  Checks if an edge exists between two given vertex ID's.
- int RemoveEdge (Vertex ∗vertex, unsigned int dest)
  Removes a specific edge from a vertex.
- int RemoveOutgoingEdges (Vertex ∗vertex)
  Removes all outgoing edges from a vertex.
- int RemoveEdgesPointingTo (Vertex ∗vertex, unsigned int targetVertexId)
  Removes edges pointing to a specific vertex from a vertex's edge list.
- int RemoveIncomingEdges (const Graph ∗graph, unsigned int vertexId)
  Removes all incoming edges to a specified vertex in the graph.

4.10.1 Descrição detalhada

Function definitions for edge creation, deletion and management.

Autor

Enrique Rodrigues

Data

22.05.2024

Copyright

4.10.2 Documentação das funções

4.10.2.1 AddEdgeToVertex()

```
bool AddEdgeToVertex (
            Vertex * vertex,
            Edge * edge )
```

Adds an edge to a vertex.

Parâmetros

vertex	- The vertex to which we add the edge.
edge	- The edge to be added.

Valores retornados

-	True in the case of success.
-	False if vertex or edge are NULL.

4.10.2.2 CreateAddEdge()

```
bool CreateAddEdge (
            Vertex * vertex,
            unsigned int dest,
            unsigned int weight )
```

Creates an edge and adds it to a vertex.

Parâmetros

vertex	- The vertex which will have the new edge.
dest	- The destination of the edge.
weight	- The weight of the edge.

Valores retornados

-	True if the edge was succesfully created and added.
-	False if edge already exists or in the event of an error.

4.10.2.3 CreateEdge()

```
Edge * CreateEdge (
            unsigned int dest,
            unsigned int weight )
```

Creates a new edge with the specified destination and weight.

Parâmetros

dest	- The destination of this edge.
weight	- The weight of this edge.

Valores retornados

-	A pointer to the newly created edge.
-	NULL if memory allocation fails.

4.10.2.4 EdgeExists()

```
bool EdgeExists (
            Vertex * vertex,
            unsigned int dest )
```

Checks if an edge exists.

Parâmetros

vertex	- The vertex where the edge starts.
dest	- The destination of the edge.

Valores retornados

-	True if the edge exists.
-	False if the edge does not exist.

4.10.2.5 EdgeExistsBetweenVertices()

```
bool EdgeExistsBetweenVertices (
            const Graph * graph,
            unsigned int src,
            unsigned int dest )
```

Checks if an edge exists between two given vertex ID's.

Parâmetros

graph	- The graph which contains the vertices and edges.
src	- The identifier of the source vertex.
dest	- The identifier of the destination vertex.

Valores retornados

-	True if the edge exists.
-	False if the edge does not exist.

4.10.2.6 RemoveEdge()

```
int RemoveEdge (
            Vertex * vertex,
            unsigned int dest )
```

Removes a specific edge from a vertex.

Parâmetros

vertex	- The vertex which contains the edge.
dest	- The destination of the edge.

Valores retornados

-	SUCCESS_REMOVING_EDGE if the edge was removed.
-	INVALID_VERTEX if the vertex is invalid.
-	EDGE_DOES_NOT_EXIST if the edge does not exist.
-	UNDEFINED_ERROR if edge was not found.

4.10.2.7 RemoveEdgesPointingTo()

```
int RemoveEdgesPointingTo (
            Vertex * vertex,
            unsigned int targetVertexId )
```

Removes edges pointing to a specific vertex from a vertex's edge list.

Parâmetros

vertex	- The vertex from which to remove edges.
target↩ VertexId	- The ID of the vertex to which the edges point.

Valores retornados

-	SUCCESS_REMOVING_EDGES if all edges were removed.
-	ERROR_REMOVING_EDGE if there was an error removing an edge.

4.10.2.8 RemoveIncomingEdges()

```
int RemoveIncomingEdges (
            const Graph * graph,
            unsigned int vertexId )
```

Removes all incoming edges to a specified vertex in the graph.

Parâmetros

graph	- The graph from which to remove incoming edges.
vertex↩ Id	- The ID of the vertex for which to remove incoming edges.

Valores retornados

-	SUCCESS_REMOVING_INCOMING_EDGES if all edges were removed.
-	INVALID_GRAPH if the graph is NULL.
-	ERROR_REMOVING_EDGE if there was an error removing an edge.

4.10.2.9 RemoveOutgoingEdges()

```
int RemoveOutgoingEdges (
            Vertex * vertex )
```

Removes all outgoing edges from a vertex.

Parâmetros

vertex	- The vertex from which to remove all outgoing edges.

Valores retornados

-	SUCCESS_REMOVING_OUTGOING_EDGES if all edges were removed.
-	VERTEX_EDGES_NULL if the vertex or its edges are NULL.
-	ERROR_REMOVING_EDGE if there was an error removing an edge.

4.11 edges.h

Ir para a documentação deste ficheiro.

```
00001
00008 #ifndef EDGES_H
00009 #define EDGES_H
00010
00011 #include <stdbool.h>
00012
00013 #include "graph.h"
00014
00022 Edge* CreateEdge(unsigned int dest, unsigned int weight);
00023
00031 bool AddEdgeToVertex(Vertex* vertex, Edge* edge);
00032
00041 bool CreateAddEdge(Vertex* vertex, unsigned int dest, unsigned int weight);
00042
00050 bool EdgeExists(Vertex* vertex, unsigned int dest);
00051
00060 bool EdgeExistsBetweenVertices(const Graph* graph, unsigned int src,
00061                                unsigned int dest);
00062
00072 int RemoveEdge(Vertex* vertex, unsigned int dest);
00073
00081 int RemoveOutgoingEdges(Vertex* vertex);
00082
00092 int RemoveEdgesPointingTo(Vertex* vertex, unsigned int targetVertexId);
00093
00102 int RemoveIncomingEdges(const Graph* graph, unsigned int vertexId);
00103
00104 #endif  // !EDGES_H
```

4.12 Referência ao ficheiro Maximal-Graph-Sum/export-graph.c

Function implementations for exporting data to a file.

```
#include "export-graph.h"
#include <stdio.h>
#include <stdlib.h>
#include "graph-structure.h"
```

Funções

- int ExportGraph (const char *filename, const Graph *graph)
  Exports a graph to a CSV file format.
- int SaveGraph (const Graph *graph, const char *filename)

4.12.1 Descrição detalhada

Function implementations for exporting data to a file.

Autor

Enrique Rodrigues

Data

21.05.2024

Copyright

4.12.2 Documentação das funções

4.12.2.1 ExportGraph()

```
int ExportGraph (
            const char * filename,
            const Graph * graph )
```

Exports a graph to a CSV file format.

Parâmetros

filename	- The name of the CSV file.
graph	- The graph to be exported.

Valores retornados

-	EXIT_SUCCESS on success
-	ERROR_OPENING_FILE if there was an error opening file

4.12.2.2 SaveGraph()

```
int SaveGraph (
            const Graph * graph,
            const char * filename )
```

Parâmetros

graph	- A pointer to the graph to be saved.
filename	- The name of the file where the graph will be saved.

Valores retornados

EXIT_SUCCESS	on success.
ERROR_OPENING_FILE	if the file cannot be opened.
ERROR_WRITING_HEADER	if there is an error writing the header.
ERROR_WRITING_VERTICES	if there is an error writing the vertices.
ERROR_WRITING_MARKER	if there is an error writing the end marker.

4.13 Referência ao ficheiro Maximal-Graph-Sum/export-graph.h

Function definitions for exporting graphs to files.

```
#include "graph.h"
```

Macros

- #define **END_MARKER** 0xFFFFFFFF
- #define **END_VERTICES_MARKER** 0xFFFFFFFE
- #define **ERROR_OPENING_FILE** -1
- #define **ERROR_WRITING_HEADER** -2
- #define **ERROR_WRITING_VERTICES** -3
- #define **ERROR_WRITING_MARKER** -4

Funções

- int ExportGraph (const char ∗filename, const Graph ∗graph)

  Exports a graph to a CSV file format.
- int SaveGraph (const Graph ∗graph, const char ∗filename)

4.13.1 Descrição detalhada

Function definitions for exporting graphs to files.

Autor

Enrique Rodrigues

Data

22.05.2024

Copyright

4.13.2 Documentação das funções

4.13.2.1 ExportGraph()

```
int ExportGraph (
            const char * filename,
            const Graph * graph )
```

Exports a graph to a CSV file format.

Parâmetros

graph	- The graph to be exported.
filename	- The name of the CSV file.

Valores retornados

-	`EXIT_SUCCESS` on success, or an error code indicating failure.

Parâmetros

filename	- The name of the CSV file.
graph	- The graph to be exported.

Valores retornados

-	`EXIT_SUCCESS` on success
-	`ERROR_OPENING_FILE` if there was an error opening file

4.13.2.2 SaveGraph()

```
int SaveGraph (
            const Graph * graph,
            const char * filename )
```

Parâmetros

graph	- A pointer to the graph to be saved.
filename	- The name of the file where the graph will be saved.

Valores retornados

EXIT_SUCCESS	on success.
ERROR_OPENING_FILE	if the file cannot be opened.
ERROR_WRITING_HEADER	if there is an error writing the header.
ERROR_WRITING_VERTICES	if there is an error writing the vertices.
ERROR_WRITING_MARKER	if there is an error writing the end marker.

4.14 export-graph.h

Ir para a documentação deste ficheiro.

```
00001
00008 #ifndef EXPORT_GRAPH_H
00009 #define EXPORT_GRAPH_H
00010
00011 #include "graph.h"
00012
00013 #define END_MARKER 0xFFFFFFFF
00014 #define END_VERTICES_MARKER 0xFFFFFFFE
00015
00016 #define ERROR_OPENING_FILE -1
00017 #define ERROR_WRITING_HEADER -2
00018 #define ERROR_WRITING_VERTICES -3
00019 #define ERROR_WRITING_MARKER -4
00020
00028 int ExportGraph(const char* filename, const Graph* graph);
00029
00039 int SaveGraph(const Graph* graph, const char* filename);
00040
00041 #endif  // !EXPORT_GRAPH_H
```

4.15 Referência ao ficheiro Maximal-Graph-Sum/graph-error-codes.h

Return code definitions which may appear from functions to the graph.

Macros

- #define **UNDEFINED_ERROR** -1
- #define **INVALID_GRAPH** -2
- #define **SUCCESS_REMOVING_EDGE** 0
- #define **SUCCESS_REMOVING_EDGES** 0
- #define **SUCCESS_REMOVING_OUTGOING_EDGES** 0
- #define **SUCCESS_REMOVING_INCOMING_EDGES** 0

- #define **EDGE_DOES_NOT_EXIST** -3
- #define **ERROR_REMOVING_EDGE** -5
- #define **SUCCESS_ADDING_VERTEX** 0
- #define **SUCCESS_REMOVING_VERTEX** 0
- #define **INVALID_VERTEX** -1
- #define **FAILURE_ADDING_VERTEX** -4
- #define **VERTEX_EDGES_NULL** -6
- #define **VERTEX_ALREADY_EXISTS** -7
- #define **VERTEX_DOES_NOT_EXIST** -8
- #define **FAILURE_CREATING_VERTEX** -9

4.15.1 Descrição detalhada

Return code definitions which may appear from functions to the graph.

Autor

Enrique Rodrigues

Data

22.05.2024

Copyright

4.16 graph-error-codes.h

Ir para a documentação deste ficheiro.

```
00001
00009 #ifndef GRAPH_ERROR_CODES_H
00010 #define GRAPH_ERROR_CODES_H
00011
00012 /* Generic return codes */
00013 #define UNDEFINED_ERROR -1
00014 #define INVALID_GRAPH -2
00015
00016 /* Edge return codes */
00017 #define SUCCESS_REMOVING_EDGE 0
00018 #define SUCCESS_REMOVING_EDGES 0
00019 #define SUCCESS_REMOVING_OUTGOING_EDGES 0
00020 #define SUCCESS_REMOVING_INCOMING_EDGES 0
00021 #define EDGE_DOES_NOT_EXIST -3
00022 #define ERROR_REMOVING_EDGE -5
00023
00024 /* Vertex return codes */
00025 #define SUCCESS_ADDING_VERTEX 0
00026 #define SUCCESS_REMOVING_VERTEX 0
00027 #define INVALID_VERTEX -1
00028 #define FAILURE_ADDING_VERTEX -4
00029 #define VERTEX_EDGES_NULL -6
00030 #define VERTEX_ALREADY_EXISTS -7
00031 #define VERTEX_DOES_NOT_EXIST -8
00032 #define FAILURE_CREATING_VERTEX -9
00033
00034 #endif  // !GRAPH_ERROR_CODES_H
```

4.17 Referência ao ficheiro Maximal-Graph-Sum/graph-structure.h

The structure definitions of the graph.

Estruturas de Dados

- struct Edge

  Structure of an edge in the graph which contains a destination vertex, weight and pointer to the next edge in the linked list.
- struct Vertex

  Structure of a vertex of a graph which contains an identification number, a linked list of all edges and a next position to traverse to the next vertices.
- struct Graph

  Structure of a graph built with a hash table for vertices and linked lists for edges.

Definições de tipos

- typedef struct Edge **Edge**
- typedef struct Vertex **Vertex**
- typedef struct Graph **Graph**

4.17.1 Descrição detalhada

The structure definitions of the graph.

Autor

Enrique Rodrigues

Data

21.05.2024

Copyright

4.18 graph-structure.h

Ir para a documentação deste ficheiro.

```
00001
00008 #ifndef GRAPH_STRUCTURE_H
00009 #define GRAPH_STRUCTURE_H
00010
00016 typedef struct Edge {
00017   unsigned int dest;    // Destination vertex
00018   unsigned int weight;  // Weight of the edge
00019   struct Edge* next;    // Pointer to the next edge in the list
00020 } Edge;
00021
00033 typedef struct Vertex {
00034   unsigned int id;      // Vertex id (identification)
00035   Edge* edges;          // Start of linked list of adjacent vertices
00036   struct Vertex* next;  // Next vertex in the hash position
00037 } Vertex;
00038
00044 typedef struct Graph {
00045   unsigned int numVertices;  // Current number of vertices of the graph
00046   unsigned int hashSize;     // Current size of hash table
00047   Vertex** vertices;         // Hash table of vertices
00048 } Graph;
00049
00050 #endif  // !GRAPH_STRUCTURE_H
```

4.19 Referência ao ficheiro Maximal-Graph-Sum/graph.c

Function implementations for standard graph functions.

```
#include "graph.h"
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
```

Funções

- Graph ∗ CreateGraph (unsigned int hashSize)

  Creates a new graph with the specified number of vertices and hash table size.
- void DisplayGraph (const Graph ∗graph)

  Displays the graph in a text format to stdout.
- void PrintEdges (const Edge ∗edge)

  Prints all edges within an edges linked list.
- void FreeGraph (Graph ∗graph)

  Frees a given graph from memory.

4.19.1 Descrição detalhada

Function implementations for standard graph functions.

Autor

Enrique Rodrigues

Data

22.05.2024

Copyright

4.19.2 Documentação das funções

4.19.2.1 CreateGraph()

```
Graph * CreateGraph (
            unsigned int hashSize )
```

Creates a new graph with the specified number of vertices and hash table size.

Parâmetros

hashSize	- The size of the hash table used to store vertices.

Valores retornados

-	A pointer to the newly created graph.
-	NULL if memory allocation fails.

4.19.2.2 DisplayGraph()

```
void DisplayGraph (
            const Graph * graph )
```

Displays the graph in a text format to stdout.

Parâmetros

graph	- The graph to be displayed.

4.19.2.3 FreeGraph()

```
void FreeGraph (
            Graph * graph )
```

Frees a given graph from memory.

Parâmetros

graph	- The graph to be freed.

4.19.2.4 PrintEdges()

```
void PrintEdges (
            const Edge * edge )
```

Prints all edges within an edges linked list.

Parâmetros

edge	- The first edge.

4.20 Referência ao ficheiro Maximal-Graph-Sum/graph.h

Main header file of a graph representation.

```
#include "graph-error-codes.h"
#include "graph-structure.h"
```

Funções

- Graph * CreateGraph (unsigned int hashSize)
  Creates a new graph with the specified number of vertices and hash table size.
- void DisplayGraph (const Graph *graph)
  Displays the graph in a text format to stdout.
- void PrintEdges (const Edge *edge)
  Prints all edges within an edges linked list.
- void FreeGraph (Graph *graph)
  Frees a given graph from memory.

4.20.1 Descrição detalhada

Main header file of a graph representation.

Includes standard graph functions such as create, display and free.

Autor

Enrique Rodrigues

Data

22.05.2024

Copyright

4.20.2 Documentação das funções

4.20.2.1 CreateGraph()

```
Graph * CreateGraph (
            unsigned int hashSize )
```

Creates a new graph with the specified number of vertices and hash table size.

Parâmetros

hashSize	- The size of the hash table used to store vertices.

Valores retornados

-	A pointer to the newly created graph.
-	NULL if memory allocation fails.

4.20.2.2 DisplayGraph()

```
void DisplayGraph (
            const Graph * graph )
```

Displays the graph in a text format to stdout.

Parâmetros

graph	- The graph to be displayed.

4.20.2.3 FreeGraph()

```
void FreeGraph (
            Graph * graph )
```

Frees a given graph from memory.

Parâmetros

graph	- The graph to be freed.

4.20.2.4 PrintEdges()

```
void PrintEdges (
            const Edge * edge )
```

Prints all edges within an edges linked list.

Parâmetros

edge	- The first edge.

4.21 graph.h

Ir para a documentação deste ficheiro.

```
00001
00010 #ifndef GRAPH_H
00011 #define GRAPH_H
00012
00013 #define WIN32_LEAN_AND_MEAN  // Exclude rarely-used stuff from Windows headers
00014
00015 #include "graph-error-codes.h"
00016 #include "graph-structure.h"
00017
00025 Graph* CreateGraph(unsigned int hashSize);
00026
00031 void DisplayGraph(const Graph* graph);
00032
00037 void PrintEdges(const Edge* edge);
00038
00043 void FreeGraph(Graph* graph);
00044
00045 #endif  // !GRAPH_H
```

4.22 Referência ao ficheiro Maximal-Graph-Sum/import-graph.c

Function implementations for importing a graph from a file.

```
#include "import-graph.h"
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "edges.h"
#include "graph.h"
#include "vertices.h"
```

Funções

- int ImportGraph (const char *filename, Graph *graph)

  Imports a graph from a text file with a CSV style format.
- Graph * LoadGraph (const char *filename)

  Loads a graph from a binary file.

4.22.1 Descrição detalhada

Function implementations for importing a graph from a file.

Autor

Enrique Rodrigues

Data

8.05.2024

Copyright

4.22.2 Documentação das funções

4.22.2.1 ImportGraph()

```
int ImportGraph (
            const char * filename,
            Graph * graph )
```

Imports a graph from a text file with a CSV style format.

Parâmetros

filename	- The name of the text file.
graph	- The graph where we will place the data.

Valores retornados

-	`SUCCESS_IMPORTING` if the graph was imported successfully.
-	`ERROR_OPENING_FILE` if the file could not be opened.
-	`MAX_FILE_SIZE_EXCEEDED` if the max file size was exceeded.
-	`ERROR_ALLOCATING_MEMORY` if there was an error allocating memory.

4.22.2.2 LoadGraph()

```
Graph * LoadGraph (
            const char * filename )
```

Loads a graph from a binary file.

Parâmetros

filename	- The name of the binary file.

Valores retornados

-	A pointer to Graph with the data inside of it or NULL in the event of an error.

4.23 Referência ao ficheiro Maximal-Graph-Sum/import-graph.h

Function definitions for importing graphs from a file.

```
#include <stdlib.h>
#include "graph.h"
```

Macros

- #define **WIN32_LEAN_AND_MEAN**
- #define **END_MARKER** 0xFFFFFFFF
- #define **END_VERTICES_MARKER** 0xFFFFFFFE
- #define **MAX_LINE_LENGTH** (1 ∗ 1024 ∗ 1024)
- #define **MAX_FILE_SIZE_MB** 200
- #define **MAX_FILE_SIZE** (MAX_FILE_SIZE_MB ∗ 1024 ∗ 1024)
- #define **ERROR_OPENING_FILE** -1
- #define **MAX_FILE_SIZE_EXCEEDED** -2
- #define **ERROR_ALLOCATING_MEMORY** -3
- #define **SUCCESS_IMPORTING** 0
- #define **INVALID_INPUT** -1

Funções

- int ImportGraph (const char ∗filename, Graph ∗graph)

  Imports a graph from a text file with a CSV style format.
- Graph ∗ LoadGraph (const char ∗filename)

  Loads a graph from a binary file.

4.23.1 Descrição detalhada

Function definitions for importing graphs from a file.

Autor

Enrique Rodrigues

Data

13.05.2024

Copyright

4.23.2 Documentação das funções

4.23.2.1 ImportGraph()

```
int ImportGraph (
            const char * filename,
            Graph * graph )
```

Imports a graph from a text file with a CSV style format.

Parâmetros

filename	- The name of the text file.
graph	- The graph where we will place the data.

Valores retornados

-	`SUCCESS_IMPORTING` if the graph was imported successfully.
-	`ERROR_OPENING_FILE` if the file could not be opened.
-	`MAX_FILE_SIZE_EXCEEDED` if the max file size was exceeded.
-	`ERROR_ALLOCATING_MEMORY` if there was an error allocating memory.

4.23.2.2 LoadGraph()

```
Graph * LoadGraph (
            const char * filename )
```

Loads a graph from a binary file.

Parâmetros

filename	- The name of the binary file.

Valores retornados

-	A pointer to Graph with the data inside of it or NULL in the event of an error.

4.24 import-graph.h

Ir para a documentação deste ficheiro.

```
00001
00008 #ifndef IMPORT_GRAPH_H
00009 #define IMPORT_GRAPH_H
00010
00011 #define WIN32_LEAN_AND_MEAN  // Exclude rarely-used stuff from Windows headers
00012
00013 #define END_MARKER 0xFFFFFFFF
00014 #define END_VERTICES_MARKER 0xFFFFFFFE
00015
00016 #define MAX_LINE_LENGTH (1 * 1024 * 1024)  // 1MB
00017 #define MAX_FILE_SIZE_MB 200
00018 #define MAX_FILE_SIZE (MAX_FILE_SIZE_MB * 1024 * 1024)
00019
00020 #define ERROR_OPENING_FILE -1
00021 #define MAX_FILE_SIZE_EXCEEDED -2
00022 #define ERROR_ALLOCATING_MEMORY -3
00023 #define SUCCESS_IMPORTING 0
00024
00025 #define INVALID_INPUT -1
00026
00027 #include <stdlib.h>
00028
00029 #include "graph.h"
00030
00043 int ImportGraph(const char* filename, Graph* graph);
00044
00051 Graph* LoadGraph(const char* filename);
00052
00053 #endif  // !IMPORT_GRAPH_H
```

4.25 Referência ao ficheiro Maximal-Graph-Sum/search.c

Function implementations for search algorithms to find all paths and calculate the sum of all edges in a path.

```
#include "search.h"
#include <stdbool.h>
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "vertices.h"
```

Funções

- bool AddPath (DFSContext ∗context)

  Adds the current path to the list of paths in the context.
- void Backtrack (DFSContext ∗context, unsigned int vertex)

  Marks the current vertex as unvisited for backtracking.
- bool TraverseEdges (DFSContext ∗context, unsigned int src, unsigned int dest)

  Recursively explores all adjacent vertices of the current vertex.
- bool DepthFirstSearch (DFSContext ∗context, unsigned int src, unsigned int dest)

  Performs Depth-First Search (DFS) on the graph from a given source to a given destination.
- PathNode ∗ FindAllPaths (const Graph ∗graph, unsigned int src, unsigned int dest, unsigned int ∗numPaths)

  Finds all paths from the source vertex to the destination vertex in the graph.
- void FreePaths (PathNode ∗paths)

  Frees the memory allocated for the list of paths.
- void PrintPaths (PathNode ∗paths)

  Prints all paths stored in the linked list of paths.
- unsigned int CalculatePathSum (const PathNode ∗path)

  Calculates the sum of the weights of the edges in the given path.

4.25.1 Descrição detalhada

Function implementations for search algorithms to find all paths and calculate the sum of all edges in a path.

Autor

Enrique Rodrigues

Data

22.05.2024

Copyright

4.25.2 Documentação das funções

4.25.2.1 AddPath()

```
bool AddPath (
            DFSContext * context )
```

Adds the current path to the list of paths in the context.

Parâmetros

context	- Pointer to the DFSContext containing the current path and paths list.

Retorna

bool - True if the path is successfully added, false otherwise.

4.25.2.2 Backtrack()

```
void Backtrack (
            DFSContext * context,
            unsigned int vertex )
```

Marks the current vertex as unvisited for backtracking.

Parâmetros

context	- Pointer to the DFSContext containing the visited status of vertices.
vertex	- The vertex to be marked as unvisited.

4.25.2.3 CalculatePathSum()

```
unsigned int CalculatePathSum (
            const PathNode * path )
```

Calculates the sum of the weights of the edges in the given path.

Parâmetros

path	- Pointer to the PathNode containing the path.

Retorna

unsigned int - The total weight of the path.

4.25.2.4 DepthFirstSearch()

```
bool DepthFirstSearch (
            DFSContext * context,
            unsigned int src,
            unsigned int dest )
```

Performs Depth-First Search (DFS) on the graph from a given source to a given destination.

Parâmetros

context	- Pointer to the DFSContext containing the graph and traversal state.
src	- The source vertex from which DFS starts.
dest	- The destination vertex to which paths are being found.

Retorna

bool - True if the DFS completes successfully, false otherwise.

4.25.2.5 FindAllPaths()

```
PathNode * FindAllPaths (
            const Graph * graph,
            unsigned int src,
            unsigned int dest,
            unsigned int * numPaths )
```

Finds all paths from the source vertex to the destination vertex in the graph.

Parâmetros

graph	- Pointer to the graph.
src	- The source vertex from which paths start.
dest	- The destination vertex to which paths are being found.
numPaths	- Pointer to store the number of paths found.

Retorna

PathNode∗ - Pointer to the head of the linked list of paths.

4.25.2.6 FreePaths()

```
void FreePaths (
            PathNode * paths )
```

Frees the memory allocated for the list of paths.

Parâmetros

paths	- Pointer to the head of the linked list of paths to be freed.

4.25.2.7 PrintPaths()

```
void PrintPaths (
            PathNode * paths )
```

Prints all paths stored in the linked list of paths.

Parâmetros

paths	- Pointer to the head of the linked list of paths to be printed.

4.25.2.8 TraverseEdges()

```
bool TraverseEdges (
            DFSContext * context,
            unsigned int src,
            unsigned int dest )
```

Recursively explores all adjacent vertices of the current vertex.

Parâmetros

context	- Pointer to the DFSContext containing the graph and traversal state.
src	- The current source vertex being explored.
dest	- The destination vertex to which paths are being found.

Retorna

bool - True if all traversals complete successfully, false otherwise.

4.26 Referência ao ficheiro Maximal-Graph-Sum/search.h

Function definitions for search algorithms to find all paths and calculate the sum of all edges in a path.

```
#include <stdbool.h>
#include "graph.h"
```

Estruturas de Dados

- struct DFSContext

  Variables that give context for the DFS function to work.
- struct PathNode

  A linked list structure which holds a path.

Definições de tipos

- typedef struct DFSContext **DFSContext**
- typedef struct PathNode **PathNode**

Funções

- bool AddPath (DFSContext ∗context)

  Adds the current path to the list of paths in the context.
- void Backtrack (DFSContext ∗context, unsigned int vertex)

  Marks the current vertex as unvisited for backtracking.
- bool TraverseEdges (DFSContext ∗context, unsigned int src, unsigned int dest)

  Recursively explores all adjacent vertices of the current vertex.
- bool DepthFirstSearch (DFSContext ∗context, unsigned int src, unsigned int dest)

  Performs Depth-First Search (DFS) on the graph from a given source to a given destination.
- PathNode ∗ FindAllPaths (const Graph ∗graph, unsigned int src, unsigned int dest, unsigned int ∗numPaths)

  Finds all paths from the source vertex to the destination vertex in the graph.
- void FreePaths (PathNode ∗paths)

  Frees the memory allocated for the list of paths.
- void PrintPaths (PathNode ∗paths)

  Prints all paths stored in the linked list of paths.
- unsigned int CalculatePathSum (const PathNode ∗path)

  Calculates the sum of the weights of the edges in the given path.

4.26.1 Descrição detalhada

Function definitions for search algorithms to find all paths and calculate the sum of all edges in a path.

Autor

Enrique Rodrigues

Data

22.05.2024

Copyright

4.26.2 Documentação das funções

4.26.2.1 AddPath()

```
bool AddPath (
            DFSContext * context )
```

Adds the current path to the list of paths in the context.

Parâmetros

context	- Pointer to the DFSContext containing the current path and paths list.

Retorna

bool - True if the path is successfully added, false otherwise.

4.26.2.2 Backtrack()

```
void Backtrack (
            DFSContext * context,
            unsigned int vertex )
```

Marks the current vertex as unvisited for backtracking.

Parâmetros

context	- Pointer to the DFSContext containing the visited status of vertices.
vertex	- The vertex to be marked as unvisited.

4.26.2.3 CalculatePathSum()

```
unsigned int CalculatePathSum (
            const PathNode * path )
```

Calculates the sum of the weights of the edges in the given path.

Parâmetros

path	- Pointer to the PathNode containing the path.

Retorna

unsigned int - The total weight of the path.

4.26.2.4 DepthFirstSearch()

```
bool DepthFirstSearch (
            DFSContext * context,
            unsigned int src,
            unsigned int dest )
```

Performs Depth-First Search (DFS) on the graph from a given source to a given destination.

Parâmetros

context	- Pointer to the DFSContext containing the graph and traversal state.
src	- The source vertex from which DFS starts.
dest	- The destination vertex to which paths are being found.

Retorna

bool - True if the DFS completes successfully, false otherwise.

4.26.2.5 FindAllPaths()

```
PathNode * FindAllPaths (
            const Graph * graph,
            unsigned int src,
            unsigned int dest,
            unsigned int * numPaths )
```

Finds all paths from the source vertex to the destination vertex in the graph.

Parâmetros

graph	- Pointer to the graph.
src	- The source vertex from which paths start.
dest	- The destination vertex to which paths are being found.
numPaths	- Pointer to store the number of paths found.

Retorna

PathNode∗ - Pointer to the head of the linked list of paths.

4.26.2.6 FreePaths()

```
void FreePaths (
            PathNode * paths )
```

Frees the memory allocated for the list of paths.

Parâmetros

paths	- Pointer to the head of the linked list of paths to be freed.

4.26.2.7 PrintPaths()

```
void PrintPaths (
            PathNode * paths )
```

Prints all paths stored in the linked list of paths.

Parâmetros

paths	- Pointer to the head of the linked list of paths to be printed.

4.26.2.8 TraverseEdges()

```
bool TraverseEdges (
            DFSContext * context,
            unsigned int src,
            unsigned int dest )
```

Recursively explores all adjacent vertices of the current vertex.

Parâmetros

context	- Pointer to the DFSContext containing the graph and traversal state.
src	- The current source vertex being explored.
dest	- The destination vertex to which paths are being found.

Retorna

bool - True if all traversals complete successfully, false otherwise.

4.27 search.h

Ir para a documentação deste ficheiro.

```
00001
00009 #ifndef SEARCH_H
00010 #define SEARCH_H
00011
00012 #include <stdbool.h>
00013
00014 #include "graph.h"
00015
00020 typedef struct DFSContext {
00021   const Graph* graph;
00022   unsigned int* pathVertices;
00023   unsigned int* pathWeights;
00024   bool* visited;
00025   unsigned int pathIndex;
00026   struct PathNode** paths;
00027   unsigned int* numPaths;
00028   unsigned int* pathCapacity;
00029 } DFSContext;
00030
00035 typedef struct PathNode {
00036   unsigned int* vertices;
00037   unsigned int* weights;
00038   unsigned int length;
00039   struct PathNode* next;
00040 } PathNode;
00041
00049 bool AddPath(DFSContext* context);
00050
00058 void Backtrack(DFSContext* context, unsigned int vertex);
00059
00069 bool TraverseEdges(DFSContext* context, unsigned int src, unsigned int dest);
00070
00081 bool DepthFirstSearch(DFSContext* context, unsigned int src, unsigned int dest);
00082
00093 PathNode* FindAllPaths(const Graph* graph, unsigned int src, unsigned int dest,
00094   unsigned int* numPaths);
00095
00101 void FreePaths(PathNode* paths);
00102
00108 void PrintPaths(PathNode* paths);
00109
00116 unsigned int CalculatePathSum(const PathNode* path);
00117
00118 #endif  // SEARCH_H
```

4.28 Referência ao ficheiro Maximal-Graph-Sum/vertices.c

Function implementations for vertex creation, deletion and management.

```
#include "vertices.h"
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>
#include "edges.h"
#include "graph.h"
```

Funções

- uint32_t Hash (uint32_t id, uint32_t hashSize)
  Computes a hash value for a given ID.
- Vertex ∗ CreateVertex (unsigned int vertexID)
  Creates a new vertex with specified vertexID.
- bool AddVertex (const Graph ∗graph, Vertex ∗vertex)
  Adds a vertex to the hash table of a graph.
- int CreateAddVertex (Graph ∗graph, unsigned int vertexID)
  Creates and adds a vertex to the hash table of a graph.
- bool VertexExists (const Graph ∗graph, unsigned int vertexID)
  Checks if a vertex exists or not.
- Vertex ∗ FindVertex (const Graph ∗graph, unsigned int vertexID)
  Tries to find a vertex from the given identifier.
- int RemoveVertex (Graph ∗graph, int vertexID)
  Removes a vertex from the graph and updates the vertex count.

4.28.1 Descrição detalhada

Function implementations for vertex creation, deletion and management.

Autor

Enrique Rodrigues

Data

22.05.2024

Copyright

4.28.2 Documentação das funções

4.28.2.1 AddVertex()

```
bool AddVertex (
            const Graph * graph,
            Vertex * vertex )
```

Adds a vertex to the hash table of a graph.

Parâmetros

graph	- The graph where the vertex should be added.
vertex	- The vertex to be added to the hash table of the graph.

Valores retornados

-	False if given graph is NULL.
-	True in the case of success.

4.28.2.2 CreateAddVertex()

```
int CreateAddVertex (
            Graph * graph,
            unsigned int vertexID )
```

Creates and adds a vertex to the hash table of a graph.

Parâmetros

graph	- The graph where the created vertex should be added.
vertexID	- The identifier of the vertex to be crated and added.

Valores retornados

-	EXIT_SUCCESS in the case of success.
-	FAILURE_CREATING_VERTEX if memory allocation fails.
-	VERTEX_ALREADY_EXISTS if the vertex already exists.

4.28.2.3 CreateVertex()

```
Vertex * CreateVertex (
            unsigned int vertexID )
```

Creates a new vertex with specified vertexID.

Parâmetros

vertexID	- The identifier of the vertex to be created.

Valores retornados

-	A pointer to the newly created vertex.
-	NULL if memory allocation fails.

4.28.2.4 FindVertex()

```
Vertex * FindVertex (
            const Graph * graph,
            unsigned int vertexID )
```

Tries to find a vertex from the given identifier.

Parâmetros

graph	- The graph which should contain the vertex.
vertexID	- The identifier of the vertex.

Valores retornados

-	A pointer to the found vertex.
-	NULL if vertex was not found.

4.28.2.5 Hash()

```
uint32_t Hash (
            uint32_t id,
            uint32_t hashSize )
```

Computes a hash value for a given ID.

This function applies a custom hashing algorithm that provides excellent statistical distribution, ensuring that each input bit influences each output bit approximately 50% of the time. Importantly, this algorithm guarantees unique outputs for distinct inputs, eliminating collisions. The algorithm is designed to be efficient, leveraging integer arithmetic and bitwise operations.

The hash function incorporates a "magic number" (0x45d9f3b), which was meticulously chosen through extensive testing. This process involved assessing the avalanche effect (the average number of output bits that change when a single input bit alters, ideally around 16), independence among output bit changes, and the likelihood of any output bit changing when any input bit changes.

The selected constant outperforms the 32-bit finalizer used by MurmurHash and approaches the quality of hashes generated by AES encryption, albeit with a slight advantage in using the same constant twice, which may offer marginal speed benefits.

The normalization of the hash value to fit within the hash table size introduces collisions, however if the hash table is big enough all collisions can be avoided.

Credit to Thomas Mueller for this algorithm: (https://stackoverflow.com/users/382763/thomas-mueller)
(https://stackoverflow.com/questions/664014/what-integer-hash-function-are-good-that-ac

Parâmetros

id	- The identifier to be hashed.
hashSize	- The size of the hash table.

Valores retornados

-	A hash value computed from input ID.

4.28.2.6 RemoveVertex()

```
int RemoveVertex (
            Graph * graph,
            int vertexID )
```

Removes a vertex from the graph and updates the vertex count.

Parâmetros

graph	- The graph which contains the vertex to be removed.
vertexID	- The ID of the vertex to be removed.

Valores retornados

-	SUCCESS_REMOVING_VERTEX if the vertex was removed.
-	INVALID_GRAPH if the graph is invalid.
-	VERTEX_DOES_NOT_EXIST if the vertex does not exist.

4.28.2.7 VertexExists()

```
bool VertexExists (
            const Graph * graph,
            unsigned int vertexID )
```

Checks if a vertex exists or not.

Parâmetros

graph	- The graph where the vertex should be.
vertexID	- The index where the vertex should be.

Valores retornados

-	True if vertex exists or False if not.

4.29 Referência ao ficheiro Maximal-Graph-Sum/vertices.h

Function definitions for vertex creation, deletion and management.

```
#include <stdbool.h>
#include <stdint.h>
```

```
#include "graph.h"
```

Macros

- #define **MIN_LOAD_FACTOR** 0.1
- #define **MAX_LOAD_FACTOR** 0.5
- #define **DEFAULT_HASH_TABLE_SIZE** 100

Funções

- uint32_t Hash (uint32_t id, uint32_t hashSize)

  Computes a hash value for a given ID.
- Vertex ∗ CreateVertex (unsigned int vertexID)

  Creates a new vertex with specified vertexID.
- bool AddVertex (const Graph ∗graph, Vertex ∗vertex)

  Adds a vertex to the hash table of a graph.
- int CreateAddVertex (Graph ∗graph, unsigned int vertexID)

  Creates and adds a vertex to the hash table of a graph.
- bool VertexExists (const Graph ∗graph, unsigned int vertexID)

  Checks if a vertex exists or not.
- Vertex ∗ FindVertex (const Graph ∗graph, unsigned int vertexID)

  Tries to find a vertex from the given identifier.
- int RemoveVertex (Graph ∗graph, int vertexID)

  Removes a vertex from the graph and updates the vertex count.

4.29.1 Descrição detalhada

Function definitions for vertex creation, deletion and management.

Autor

Enrique Rodrigues

Data

22.05.2024

Copyright

4.29.2 Documentação das funções

4.29.2.1 AddVertex()

```
bool AddVertex (
            const Graph * graph,
            Vertex * vertex )
```

Adds a vertex to the hash table of a graph.

Parâmetros

graph	- The graph where the vertex should be added.
vertex	- The vertex to be added to the hash table of the graph.

Valores retornados

-	False if given graph is NULL.
-	True in the case of success.

4.29.2.2 CreateAddVertex()

```
int CreateAddVertex (
            Graph * graph,
            unsigned int vertexID )
```

Creates and adds a vertex to the hash table of a graph.

Parâmetros

graph	- The graph where the created vertex should be added.
vertexID	- The identifier of the vertex to be crated and added.

Valores retornados

-	EXIT_SUCCESS in the case of success.
-	FAILURE_CREATING_VERTEX if memory allocation fails.
-	VERTEX_ALREADY_EXISTS if the vertex already exists.

4.29.2.3 CreateVertex()

```
Vertex * CreateVertex (
            unsigned int vertexID )
```

Creates a new vertex with specified vertexID.

Parâmetros

vertexID	- The identifier of the vertex to be created.

Valores retornados

-	A pointer to the newly created vertex. NULL if memory allocation fails.

Parâmetros

Parâmetros

vertexID	- The identifier of the vertex to be created.

Valores retornados

-	A pointer to the newly created vertex.
-	NULL if memory allocation fails.

4.29.2.4 FindVertex()

```
Vertex * FindVertex (
            const Graph * graph,
            unsigned int vertexID )
```

Tries to find a vertex from the given identifier.

Parâmetros

graph	- The graph which should contain the vertex.
vertexID	- The identifier of the vertex.

Valores retornados

-	A pointer to the found vertex.
-	NULL if vertex was not found.

4.29.2.5 Hash()

```
uint32_t Hash (
            uint32_t id,
            uint32_t hashSize )
```

Computes a hash value for a given ID.

Parâmetros

id	- The identifier to be hashed.
hashSize	- The size of the hash table.

Valores retornados

-	A hash value computed from input ID.

This function applies a custom hashing algorithm that provides excellent statistical distribution, ensuring that each input bit influences each output bit approximately 50% of the time. Importantly, this algorithm guarantees unique outputs for distinct inputs, eliminating collisions. The algorithm is designed to be efficient, leveraging integer arithmetic and bitwise operations.

The hash function incorporates a "magic number" (0x45d9f3b), which was meticulously chosen through extensive testing. This process involved assessing the avalanche effect (the average number of output bits that change when a single input bit alters, ideally around 16), independence among output bit changes, and the likelihood of any output bit changing when any input bit changes.

The selected constant outperforms the 32-bit finalizer used by MurmurHash and approaches the quality of hashes generated by AES encryption, albeit with a slight advantage in using the same constant twice, which may offer marginal speed benefits.

The normalization of the hash value to fit within the hash table size introduces collisions, however if the hash table is big enough all collisions can be avoided.

Credit to Thomas Mueller for this algorithm: (https://stackoverflow.com/users/382763/thomas-mueller)
(https://stackoverflow.com/questions/664014/what-integer-hash-function-are-good-that-ac

Parâmetros

id	- The identifier to be hashed.
hashSize	- The size of the hash table.

Valores retornados

-	A hash value computed from input ID.

4.29.2.6 RemoveVertex()

```
int RemoveVertex (
            Graph * graph,
            int vertexID )
```

Removes a vertex from the graph and updates the vertex count.

Parâmetros

graph	- The graph which contains the vertex to be removed.
vertexID	- The ID of the vertex to be removed.

Valores retornados

-	SUCCESS_REMOVING_VERTEX if the vertex was removed.
-	INVALID_GRAPH if the graph is invalid.
-	VERTEX_DOES_NOT_EXIST if the vertex does not exist.

4.29.2.7 VertexExists()

```
bool VertexExists (
            const Graph * graph,
            unsigned int vertexID )
```

Checks if a vertex exists or not.

Parâmetros

graph	- The graph where the vertex should be.
vertexID	- The index where the vertex should be.

Valores retornados

-	True if vertex exists or False if not.

4.30 vertices.h

Ir para a documentação deste ficheiro.

```
00001
00009 #ifndef VERTICES_H
00010 #define VERTICES_H
00011
00012 #include <stdbool.h>
00013 #include <stdint.h>
00014
00015 #include "graph.h"
00016
00017 #define MIN_LOAD_FACTOR 0.1
00018 #define MAX_LOAD_FACTOR 0.5
00019
00020 #define DEFAULT_HASH_TABLE_SIZE 100
00021
00029 uint32_t Hash(uint32_t id, uint32_t hashSize);
00030
00037 Vertex* CreateVertex(unsigned int vertexID);
00038
00046 bool AddVertex(const Graph* graph, Vertex* vertex);
00047
00056 int CreateAddVertex(Graph* graph, unsigned int vertexID);
00057
00064 bool VertexExists(const Graph* graph, unsigned int vertexID);
00065
00073 Vertex* FindVertex(const Graph* graph, unsigned int vertexID);
00074
00083 int RemoveVertex(Graph* graph, int vertexID);
00084
00085 #endif  // !VERTICES_H
```

Índice