CPPDescent

80d9539 (with uncommitted changes)

Generated by Doxygen 1.9.7

Chapter 1

CPP-Descent

Part of the "Software Development for Computing Systems" course of the Department of Informatics and Telecommunications, UoA for the first semester of the academic year 2023-2024.

A library that implements the K-NN graph creation algorithm described in [1], [2], [4].

1.1 Contributors

- Konstantinos Chousos (1115202000215, sdi2000215 *at* di.uoa.gr)

  Contributions

  - 1st submission
    - Build/Testing/Code coverage pipeline
    - ADTVector, ADTList, ADTPQueue
    - Brute Force K-NN Graph creation. NN-Descent Graph creation
    - (Incomplete) Query point
  - 2nd submission
    - I/O of graphs in binary files
    - Recall function
    - Metric functions
    - Query point
    - Early termination
    - Local join
  - 3rd submission
    - Incremental search
    - Sampling
    - Code refactoring, removing Map data structure from the graph
    - Incorporation of the GNU Scientific Library [7] for faster mathematical computations
    - Optimization of the distance metric, by using linear computations
    - Implementation of CLI flags for the main app
    - Google Benchmark integration (DEFUNCT)
    - Refactoring of the lookup of a datapoint's neighbors, leading to major speed up
    - Script for experiments that outputs a csv file with the results
    - Recursive random projection trees [6], [5]

- Anastasios-Fedon Seitanidis (1115202000179, sdi2000179 *at* di.uoa.gr)

  Contributions

  - 1st submission
    - ADTMap, ADTGraph
    - Brute Force K-NN Graph creation
    - NN-Descent Graph creation
  - 2nd submission
    - Implementation of GraphVertex class
    - Refactor of previous Graph implementation to use GraphVertex class (without changing the existing interface)
    - `getMin()` property to Priority Queue
    - `remove()` property to Priority Queue
    - Implementation of `find()` helper function for Priority Queue

1.2 Running the project

To run the main executable, the following commands suffice:

```
$ cmake -S . -B build
$ cmake --build build
$ ./build/app/app -K <number of neighbors> <filepath-to-dataset>
```

For example:

```
$ ./build/app/app -K 100 ./datasets/00000200.bin -D 20 -T 5 -d 0.001 -r 0.3
For K = 100
For  = 0.001
For  = 0.3
5 random projection trees are used, for D = 20
Dataset: ./datasets/00000200.bin
Dimensions: 100
-------------------------------------------------------------
NN-Descent
        Initializing starting graph...
        Using random projection tree...
        Starting graph has been created
        Number of changes in the graph (c) = 1950
        Number of changes in the graph (c) = 540
        Number of changes in the graph (c) = 84
        Number of changes in the graph (c) = 36
        Number of changes in the graph (c) = 15
        NN-Descent iterations: 5
NN-Descent K-NN Graph created in 3007 milliseconds for  = 0.001,  = 0.3
Computing recall...
Total recall is 98.9451%
```

1.2.1 Command-line flags

Flag	Default value	Usage
`-K`	-	The K number of neighbors for each datapoint. This value *must* be given.
`-d`	0.01	The precision parameter δ used for early termination. If the updates during an iteration are less than δKN, then the algorithm concludes.
`-r`	0.5	The sample rate ρ used for sampling. Before local join, we sample ρK out of the K-NN items marked `true` for each object to use in local join.
`-D`	0	The upper bound of vertices a *random projection tree*'s leaf must have. If set to 0, then the starting graph is created randomly. Must be less than K.
`-T`	4	The number of *random projection trees* to create. Must be at least 2.
`-q`		For "quiet". If this flag is set, then the output is a line of comma separated values.

1.2.2 Results

The *recall* percentage that is printed during execution is computed by the following formula, derived from [1, Sec. 3.2].

> The recall of one object is the number of its true K-NN members found divided by K. The recall of an approximate K-NNG is the average recall of all objects.

1.3 Documentation

1.3.1 Code Structure

In the wiki, you can find documentation generated by Doxygen. It is also available in pdf form.

1.3.2 Code Coverage

This project uses LCOV for its code coverage needs. The results are uploaded upon commit here.

1.4 Dependencies

For local development, you will need Make and CMake. To install them on an apt-based linux distro, run the following commands:

```
$ sudo apt update && sudo apt upgrade
$ sudo apt install make
$ sudo apt install cmake
```

1.4.1 Minor dependencies

- The project uses Google Test for its testing needs. Gtest is used as a git submodule of this repo, that is downloaded by CMake the first time you clone. That means that nothing extra needs to be installed to run the tests.
- If you would like to generate the documentation, you will need doxygen installed.

1.5 Bibliography

[1] W. Dong, C. Moses, and K. Li, “Efficient k-nearest neighbor graph construction for generic similarity measures,” in Proceedings of the 20th international conference on World wide web, in WWW ’11. New York, NY, USA: Association for Computing Machinery, Mar. 2011, pp. 577–586. doi: 10.1145/1963405.1963487.

[2] “How PyNNDescent works — pynndescent 0.5.0 documentation.” Accessed: Oct. 10, 2023. [Online]. Available: https://pynndescent.readthedocs.io/en/latest/how_pynndescent_works.html

[3] D. Kluser, J. Bokstaller, S. Rutz, and T. Buner, “Fast Single-Core K-Nearest Neighbor Graph Computation.” arXiv, Dec. 13, 2021. doi: 10.48550/arXiv.2112.06630.

[4] PyNNDescent Fast Approximate Nearest Neighbor Search with Numba | SciPy 2021, (Jul. 23, 2021). Accessed: Dec. 05, 2023. [Online Video]. Available: https://www.youtube.com/watch?v=xPadY4_kt3o

[5] J. Brugger, “brj0/nndescent.” Dec. 11, 2023. Accessed: Dec. 11, 2023. [Online]. Available: https://github.com/brj0/nndescent

[6] S. Dasgupta and Y. Freund, “Random projection trees and low dimensional manifolds,” in Proceedings of the fortieth annual ACM symposium on Theory of computing, Victoria British Columbia Canada: ACM, May 2008, pp. 537–546. doi: 10.1145/1374376.1374452.

[7] “GSL - GNU Scientific Library - GNU Project - Free Software Foundation.” Accessed: Jan. 15, 2024. [Online]. Available: https://www.gnu.org/software/gsl/

Chapter 2

Namespace Index

2.1 Namespace List

Here is a list of all documented namespaces with brief descriptions:

Chapter 3

Class Index

3.1 Class List

Here are the classes, structs, unions and interfaces with brief descriptions:

Chapter 4

File Index

4.1 File List

Here is a list of all documented files with brief descriptions:

Chapter 5

Namespace Documentation

5.1 cppdescent Namespace Reference

Functions for the creation of a K-NN graph.

Functions

- int **compareGraphVertices** (Pointer vertex1, Pointer vertex2)
- int **compareGraphVertexPairs** (Pointer p1, Pointer p2)
- Vector * readBinData (const char *fp, int dimensions)

  Reads the data from a binary file.
- void writeBinGraph (const char *fp, Graph *graph, int K)

  Writes a computed graph to a binary file.
- Graph * readBinGraph (const char *fp, int dimensions)

  Reads a graph from a binary file. To work correctly, the file needs to be first created from the 'writeBinGraph' function.
- float recall (Graph *bfGraph, Graph *nnGraph, int N, int K)

  Returns the recall of the graph computed by NN-Descent, compared to the brute force graph.
- float euclideanDistance (Pointer a, Pointer b)

  Returns the Euclidean distance between two points of arbitrary dimension.
- float manhattanDistance (Pointer a, Pointer b)

  Returns the Manhattan distance between two points of arbitrary dimension.
- int compareEdgesEuclidean (Pointer first, Pointer second)

  Compare edges using the euclideanDistance function.
- int **compareVertexPairsEuclidean** (Pointer first, Pointer second)
- int compareEdgesManhattan (Pointer first, Pointer second)

  Compare edges using the manhattanDistance function.
- Graph * KNNBruteForceGraph (Vector *data, int K, CompareFunc compare)

  Computes the K-NN graph using brute force.
- Graph * NNDescent_KNNGraph (Vector *data, int K, int D, int Trees, float delta, float rho, DistanceFunc distance)

  Computes the K-NN graph for the given dataset using the NN-Descent algorithm.
- PQueue * NNDescent_Query (Graph *graph, int K, CompareFunc compare, Vector *query)

  Computes the K Nearest Neighbors of the query point in the graph.
- void RPTree (Graph *graph, Vector *vec, int K, int D, int dimensions)

  Creates a graph using random projection trees.

5.1.1 Detailed Description

Functions for the creation of a K-NN graph.

5.1.2 Function Documentation

5.1.2.1 compareEdgesEuclidean()

```
int cppdescent::compareEdgesEuclidean (
            Pointer first,
            Pointer second )
```

Compare edges using the euclideanDistance function.

Parameters

first	A Pointer to the first element.
second	A Pointer to the second element.

Returns

int

5.1.2.2 compareEdgesManhattan()

```
int cppdescent::compareEdgesManhattan (
            Pointer first,
            Pointer second )
```

Compare edges using the manhattanDistance function.

Parameters

first	A Pointer to the first element.
second	A Pointer to the second element.

Returns

int

5.1.2.3 euclideanDistance()

```
float cppdescent::euclideanDistance (
            Pointer a,
            Pointer b )
```

Returns the Euclidean distance between two points of arbitrary dimension.

Parameters

first	A pointer to the first point.
second	A pointer to the second point.

Returns

long double The Euclidean distance.

5.1.2.4 KNNBruteForceGraph()

```
Graph * cppdescent::KNNBruteForceGraph (
            Vector * data,
            int K,
            CompareFunc compare )
```

Computes the K-NN graph using brute force.

Useful for testing the recall and the performance of our NN-Descent algorithm.

The user is responsible deletion of the allocated memory of the graph.

Parameters

data	A pointer to the parent N-sized vector.
K	The number of Nearest Neigbors to find.
compare	The function to use to compare the distances.

Returns

Graph* A pointer to the optimal K-NN graph.

5.1.2.5 manhattanDistance()

```
float cppdescent::manhattanDistance (
            Pointer a,
            Pointer b )
```

Returns the Manhattan distance between two points of arbitrary dimension.

Parameters

first	A pointer to the first point.
second	A pointer to the second point.

Returns

long double The Manhattan distance.

5.1.2.6 NNDescent_KNNGraph()

```
Graph * cppdescent::NNDescent_KNNGraph (
            Vector * data,
            int K,
            int D,
            int Trees,
            float delta,
            float rho,
            DistanceFunc distance )
```

Computes the K-NN graph for the given dataset using the NN-Descent algorithm.

The user is responsible deletion of the allocated memory of the returned graph.

Parameters

data	A Vector of the vertices of the graph.
K	The number of nearest neighbors to compute.
K	The number of nearest neighbors to compute.
K	The number of nearest neighbors to compute.
delta	The iterations will stop when the number of edges that were updated is less than delta∗N∗K.
rho	The sampling rate
distance	The function to be used to compute the distances between vertices.

Returns

Graph∗ The complete K-NN graph of the dataset.

5.1.2.7 NNDescent_Query()

```
PQueue * cppdescent::NNDescent_Query (
            Graph * graph,
            int K,
            CompareFunc compare,
            Vector * query )
```

Computes the K Nearest Neighbors of the query point in the graph.

Parameters

graph	The K-NN Graph, in which we search for neighbors of the query.
K	
compare	The function used to compare edges.
query	The query point, given as a Vector. Must be of the same dimensions as the rest points of the graph.

Returns

PQueue∗ A priority queue of the K-NN of the query, with 'max' being the most distant point relative to the query. It contains GraphVertexPairs from the query point directed to the other nodes. If the query point is of wrong dimensions, then nullptr is returned.

5.1.2.8 readBinData()

```
Vector * cppdescent::readBinData (
            const char * fp,
            int dimensions )
```

Reads the data from a binary file.

The data read are stored in a N-sized Vector, where N is the <uint32_t> at the start of the file. Each node of this vector is a pointer to another 100-sized vector, of which the values are pointers to floats.

All of the vectors (the parent N-sized and each of the 100-sized) must be freed by the user.

Parameters

fp	A string (char *) of the filepath to the binary dataset.
dimensions	The dimension of each point.

Returns

Vector* An N-sized vector with vectors for each element.

5.1.2.9 readBinGraph()

```
Graph * cppdescent::readBinGraph (
            const char * fp,
            int dimensions )
```

Reads a graph from a binary file. To work correctly, the file needs to be first created from the 'writeBinGraph' function.

Parameters

fp	The filepath.
dimensions	The dimensions of the datapoints.

Returns

Graph*

5.1.2.10 recall()

```
float cppdescent::recall (
            Graph * bfGraph,
            Graph * nnGraph,
            int N,
            int K )
```

Returns the recall of the graph computed by NN-Descent, compared to the brute force graph.

Parameters

bfGraph	
nnGraph	
N	
K	

Returns

float

5.1.2.11 RPTree()

```
void cppdescent::RPTree (
            Graph * graph,
            Vector * vec,
            int K,
            int D,
            int dimensions )
```

Creates a graph using random projection trees.

Starts by splitting the dataspace using a hyperplane, and calling itself recursively in each ¨slice¨.

The starting graph has no edges, only the vertices.

Parameters

graph	
vec	
K	The open upper bound of elements in a leaf.
D	The maximum number of elements in a leaf. Must be less than K to result in a connected graph.
dimensions	The dimensions of each datapoint.

5.1.2.12 writeBinGraph()

```
void cppdescent::writeBinGraph (
            const char * fp,
            Graph * graph,
            int K )
```

Writes a computed graph to a binary file.

The structure of the binary file that will be created is the following:

1. <uint32_t> N: the number of the vertices
2. N ∗ 100 floats: each vertex of 100 dimensions
3. N ∗ K ∗ 1 int: For each vertex sequentially, the positions of its K neighbors.

Parameters

fp	The filepath to the created file.
K	
graph	

Chapter 6

Class Documentation

6.1 Graph Class Reference

Public Member Functions

- **Graph** (CompareFunc compare, DestroyFunc vecDestroy)
- int **getSize** ()
- void insertVertex (Pointer vertex)

  Insert a vertex to the graph (if it doesn't already exist).
- Vector * **getVerticesV** ()
- void **removeVertex** (Pointer vertex)
- void **insertEdge** (Pointer vertex1, Pointer vertex2)
- void **removeEdge** (Pointer vertex1, Pointer vertex2)
- Vector * **getAdjacentV** (Pointer vertex)
- Vector * **getReverseAdjacentV** (Pointer vertex)
- Vector * getGeneralNeighborsV (Pointer vertex)

  Returns a vector where the K first elements are the direct neighbors of the vertex and the rest are the reverse ones.
- bool **isNeighborVertex** (Pointer v1, Pointer v2)
- CompareFunc **getCompareData** ()
- CompareFunc **getCompareVertices** ()
- DestroyFunc **getDestroyData** ()
- Vector * **getVec** ()

6.1.1 Member Function Documentation

6.1.1.1 getGeneralNeighborsV()

```
Vector * Graph::getGeneralNeighborsV (
            Pointer vertex )
```

Returns a vector where the K first elements are the direct neighbors of the vertex and the rest are the reverse ones.

Parameters

vertex	

Returns

Vector∗

6.1.1.2 insertVertex()

```
void Graph::insertVertex (
            Pointer vertex )
```

Insert a vertex to the graph (if it doesn't already exist).

Parameters

vertex	A Pointer to the vertex to be unserted.

The documentation for this class was generated from the following files:

- ADTGraph.hpp
- ADTGraph.cpp

6.2 GraphVertex Class Reference

Public Member Functions

- **GraphVertex** (Pointer data, Graph ∗owner)
- void **setPos** (int pos)
- int **getPos** ()
- void **addNeighbor** (Pointer neighbor)
- void **addReverse** (Pointer reverse)
- void **removeNeighbor** (Pointer neighbor, CompareFunc compare)
- void **removeReverse** (Pointer reverse, CompareFunc compare)
- PQueue ∗ **getNeighbors** ()
- PQueue ∗ **getReverse** ()
- Pointer **getData** ()
- Graph ∗ **getOwner** ()
- void **check** ()
- bool **checked** ()
- void **setNorm** (double norm)
- double **getNorm** ()

The documentation for this class was generated from the following files:

- ADTGraph.hpp
- ADTGraph.cpp

6.3 GraphVertexPair Class Reference

Public Member Functions

- **GraphVertexPair** (Graph ∗owner, Pointer vertex1, Pointer vertex2)
- Pointer **getVertex1** ()
- Pointer **getVertex2** ()
- Graph ∗ **getOwner** ()
- void **setFalse** ()
- bool **getFlag** ()

The documentation for this class was generated from the following file:

- ADTGraph.hpp

6.4 PQueue Class Reference

ADT Priority Queue.

`#include <ADTPQueue.hpp>`

Public Member Functions

- PQueue (CompareFunc compare, DestroyFunc destroyValue, Vector ∗values)

  Construct a new PQueue object.
- ∼PQueue ()

  Destroy the PQueue object.
- int getSize ()

  Get the size of the priority queue.
- Pointer getMax ()

  Get the max element of the queue.
- Pointer getMin ()

  Get the min element of the queue.
- void insert (Pointer value)

  Insert a new element to the queue.
- void removeMax ()

  Remove the max of the queue.
- DestroyFunc setDestroyValue (DestroyFunc destroyValue)

  Set the Destroy Value object.
- void remove (Pointer value, CompareFunc compare)

  Finds (using the given compare function) and removes the node with the given value.
- int find (Pointer value, CompareFunc compare)

  Finds the given value in the priority queue and returns its nodeId.
- Vector ∗ toVector ()

  Returns all the elements as a vector.
- Pointer **nodeValue** (int nodeId)
- void **nodeSwap** (int nodeId1, int nodeId2)
- void **bubbleUp** (int nodeId)
- void **bubbleDown** (int nodeId)
- void **naiveHeapify** (Vector ∗values)

6.4.1 Detailed Description

ADT Priority Queue.

6.4.2 Constructor & Destructor Documentation

6.4.2.1 PQueue()

```
PQueue::PQueue (
            CompareFunc compare,
            DestroyFunc destroyValue,
            Vector * values )
```

Construct a new PQueue object.

Uses the `compare` function to compare its elements.

If `destroyValue` != nullptr, the `destroyValue` function is called upon removal of an element.

If `values` != nullptr, then the priority queue will be initialized with the values of the vector `values`.

Parameters

compare	
destroyValue	
values	

6.4.2.2 ~PQueue()

```
PQueue::~PQueue ( )
```

Destroy the PQueue object.

6.4.3 Member Function Documentation

6.4.3.1 find()

```
int PQueue::find (
            Pointer value,
            CompareFunc compare )
```

Finds the given value in the priority queue and returns its nodeId.

Parameters

value	The value we want to find.
compare	Compare function to use.

Returns

int

6.4.3.2 getMax()

```
Pointer PQueue::getMax ( )
```

Get the max element of the queue.

Returns

Pointer A generic pointer to the max element.

6.4.3.3 getMin()

```
Pointer PQueue::getMin ( )
```

Get the min element of the queue.

Returns

Pointer A generic pointer to the min element.

6.4.3.4 getSize()

```
int PQueue::getSize ( )
```

Get the size of the priority queue.

Returns

int

6.4.3.5 insert()

```
void PQueue::insert (
            Pointer value )
```

Insert a new element to the queue.

Parameters

value	The value of the new element to be inserted.

6.4.3.6 remove()

```
void PQueue::remove (
            Pointer value,
            CompareFunc compare )
```

Finds (using the given compare function) and removes the node with the given value.

Parameters

value	The value we want to remove.
compare	Compare function to use for finding the node we want to remove.

6.4.3.7 removeMax()

```
void PQueue::removeMax ( )
```

Remove the max of the queue.

6.4.3.8 setDestroyValue()

```
DestroyFunc PQueue::setDestroyValue (
            DestroyFunc destroyValue )
```

Set the Destroy Value object.

Parameters

destroyValue	

Returns

DestroyFunc

6.4.3.9 toVector()

```
Vector * PQueue::toVector ( )  [inline]
```

Returns all the elements as a vector.

Returns

Vector∗

The documentation for this class was generated from the following files:

- ADTPQueue.hpp
- ADTPQueue.cpp

6.5 sets Struct Reference

Collaboration diagram for sets:

Public Attributes

- Vector * **new_v**
- Vector * **old_v**

The documentation for this struct was generated from the following file:

- cppdescent.cpp

6.6 Vector Class Reference

ADT Vector.

```
#include <ADTVector.hpp>
```

Public Member Functions

- Vector (int size, DestroyFunc destroyValue)

  Construct a new Vector object.
- ~Vector ()

  Destroy the Vector object.
- int getSize ()

  Get the current size of the Vector.
- void insertLast (Pointer value)

  Inserts a new element at the end of the vector.
- int removeLast ()

  Removes the last element of the vector.
- Pointer getAt (int pos)

  Get the value at the specified position of the vector.

- int setAt (int pos, Pointer value)

  Set the value at the specified position of the vector.
- Pointer find (Pointer value, CompareFunc compare)

  Find the first element with value equal to value.
- Pointer binaryFind (Pointer value, CompareFunc compare)

  Find the element with value equal to value using binary search.
- int findPos (Pointer value, CompareFunc compare)

  Find the first element with value equal to value and return its pos.
- DestroyFunc setDestroyValue (DestroyFunc destroyValue)

  Set the Destroy Value.
- vectorNode * first ()

  Get the first node of the vector.
- vectorNode * last ()

  Get the last node of the vector.
- vectorNode * next (vectorNode *node)

  Get the next node of the vector, after the one given.
- vectorNode * previous (vectorNode *node)

  Get the previous node of the vector, before the one given.
- Pointer nodeValue (vectorNode *node)

  Get the value of the node.
- vectorNode * findNode (Pointer value, CompareFunc compare)

  Find the first node with value equal to value.
- void swap (int pos1, int pos2)

  Swaps the value that is saved in pos1 with the one in pos2.

6.6.1 Detailed Description

ADT Vector.

An ADT Vector using a dynamic array for smart resizing.

6.6.2 Constructor & Destructor Documentation

6.6.2.1 Vector()

```
Vector::Vector (
            int size,
            DestroyFunc destroyValue )
```

Construct a new Vector object.

The newly created Vector will be of size size and its elements will be initialized as nullptr.

If destroyValue is not nullptr, it will be called on each element when it is removed from the Vector.

Parameters

size	The initial size of the Vector.
destroyValue	The function to be called on each element when it is removed from the Vector.

6.6.2.2 ~Vector()

```
Vector::~Vector ( )
```

Destroy the Vector object.

Deletes all allocated memory of the Vector.

6.6.3 Member Function Documentation

6.6.3.1 binaryFind()

```
Pointer Vector::binaryFind (
            Pointer value,
            CompareFunc compare )
```

Find the element with value equal to value using binary search.

Parameters

value	The value to look for.
compare	The function to be used for comparison.

Returns

Pointer A pointer to the found value.

6.6.3.2 find()

```
Pointer Vector::find (
            Pointer value,
            CompareFunc compare )
```

Find the first element with value equal to value.

Parameters

value	The value to look for.
compare	The function to be used for comparison.

Returns

Pointer A pointer to the found value.

6.6.3.3 findNode()

```
vectorNode * Vector::findNode (
            Pointer value,
            CompareFunc compare )
```

Find the first node with value equal to value.

Parameters

value	The value to look for.
compare	The function to be used for comparison.

Returns

vectorNode The resulting node.

6.6.3.4 findPos()

```
int Vector::findPos (
            Pointer value,
            CompareFunc compare )
```

Find the first element with value equal to value and return its pos.

Parameters

value	The value to look for.
compare	The function to be used for comparison.

Returns

int The position of the element.

6.6.3.5 first()

```
vectorNode * Vector::first ( )
```

Get the first node of the vector.

Returns

vectorNode

6.6.3.6 getAt()

```
Pointer Vector::getAt (
            int pos )
```

Get the value at the specified position of the vector.

Parameters

pos	The position to look for.

Returns

Pointer A pointer to the value at the specified position.

6.6.3.7 getSize()

```
int Vector::getSize ( )
```

Get the current size of the Vector.

Returns

int The size of the Vector.

6.6.3.8 insertLast()

```
void Vector::insertLast (
            Pointer value )
```

Inserts a new element at the end of the vector.

Parameters

value	The value of the new element.

If we are at the last element, we create a new array double the size.

6.6.3.9 last()

```
vectorNode * Vector::last ( )
```

Get the last node of the vector.

Returns

vectorNode

6.6.3.10 next()

```
vectorNode * Vector::next (
            vectorNode * node )
```

Get the next node of the vector, after the one given.

Parameters

node	The node to get the next of.

Returns

vectorNode

6.6.3.11 nodeValue()

```
Pointer Vector::nodeValue (
            vectorNode * node )
```

Get the value of the node.

Parameters

node	

Returns

Pointer

6.6.3.12 previous()

```
vectorNode * Vector::previous (
            vectorNode * node )
```

Get the previous node of the vector, before the one given.

Parameters

node	The node to get the previous of.

Returns

vectorNode

6.6.3.13 removeLast()

```
int Vector::removeLast ( )
```

Removes the last element of the vector.

Returns

int -1 if the function fails, else 0.

6.6.3.14 setAt()

```
int Vector::setAt (
            int pos,
            Pointer value )
```

Set the value at the specified position of the vector.

Parameters

pos	The position to edit.
value	The new value.

Returns

int -1 if the function fails, else 0.

6.6.3.15 setDestroyValue()

```
DestroyFunc Vector::setDestroyValue (
            DestroyFunc destroyValue )
```

Set the Destroy Value.

Parameters

destroyValue	The new destroy value function.

Returns

DestroyFunc The old destroy value function.

6.6.3.16 swap()

```
void Vector::swap (
            int pos1,
            int pos2 )
```

Swaps the value that is saved in pos1 with the one in pos2.

Parameters

pos1	Index of the first element to be swaped.
pos2	Index of the second element to be swaped.

Returns

vectorNode The resulting node.

The documentation for this class was generated from the following files:

- ADTVector.hpp
- ADTVector.cpp

6.7 vectorNode Class Reference

Class for the vector Node object.

`#include <ADTVector.hpp>`

Public Member Functions

- vectorNode ()
  Construct a new vector Node object.
- **vectorNode** (Pointer value)
- Pointer getValue () const
  Get the Value object.
- void setValue (Pointer value)
  Set the Value object.

6.7.1 Detailed Description

Class for the vector Node object.

A vector node is described only its value.

6.7.2 Constructor & Destructor Documentation

6.7.2.1 vectorNode()

`vectorNode::vectorNode ( )  [inline]`

Construct a new vector Node object.

Parameters

value	A Pointer to the value.

6.7.3 Member Function Documentation

6.7.3.1 getValue()

```
Pointer vectorNode::getValue ( ) const  [inline]
```

Get the Value object.

Returns

Pointer

6.7.3.2 setValue()

```
void vectorNode::setValue (
            Pointer value )  [inline]
```

Set the Value object.

Parameters

value	

The documentation for this class was generated from the following file:

- ADTVector.hpp

Chapter 7

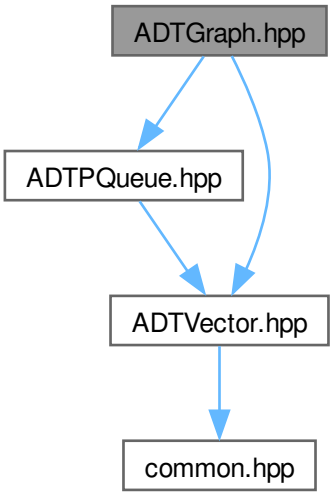
File Documentation

7.1 ADTGraph.hpp File Reference

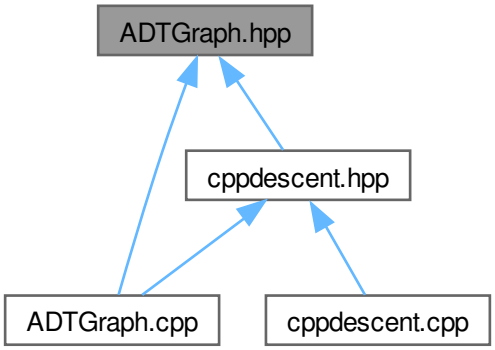
Abstract undirected graph with weighted edges.

```
#include ¨ADTPQueue.hpp¨
#include ¨ADTVector.hpp¨
```

Include dependency graph for ADTGraph.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class Graph
- class GraphVertex
- class GraphVertexPair

7.1.1 Detailed Description

Abstract undirected graph with weighted edges.

Author

Phaedon Seitanidis

Version

0.1

Date

2023-11-01

Copyright

Copyright (c) 2023

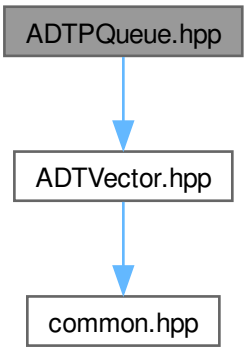
7.2 ADTGraph.hpp

Go to the documentation of this file.

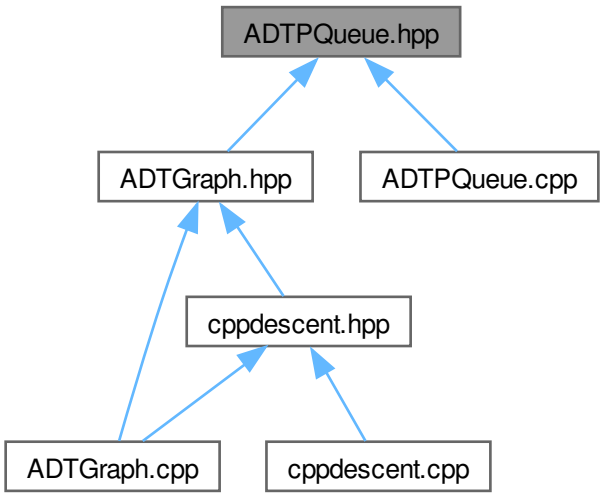
```
00001
00012 #pragma once
00013
00014 #include "ADTPQueue.hpp"
00015 #include "ADTVector.hpp"
00016
00017 class Graph {
00018  private:
00019   Vector* vec;
00020   int size;
00021   CompareFunc compare_vertices;
00022   CompareFunc compare_data;
00023   DestroyFunc destroy_data;
00024
00025  public:
00026   Graph(CompareFunc compare, DestroyFunc vecDestroy);
00027   ~Graph();
00028   int getSize();
00029   void insertVertex(Pointer vertex);
00030   Vector* getVerticesV();
00031   void removeVertex(Pointer vertex);
00032   void insertEdge(Pointer vertex1, Pointer vertex2);
00033   void removeEdge(Pointer vertex1, Pointer vertex2);
00034   Vector* getAdjacentV(Pointer vertex);
00035   Vector* getReverseAdjacentV(Pointer vertex);
00036   Vector* getGeneralNeighborsV(Pointer vertex);
00037   bool isNeighborVertex(Pointer v1, Pointer v2);
00038   CompareFunc getCompareData() { return this->compare_data; };
00039   CompareFunc getCompareVertices() { return this->compare_vertices; };
00040   DestroyFunc getDestroyData() { return this->destroy_data; };
00041   Vector* getVec() { return this->vec; };
00042 };
00043
00044 class GraphVertex {
00045  private:
00046   Pointer data;
00047   PQueue* neighbors;
00048   PQueue* reverse;
00049   Graph* owner;
00050   bool hasBeenChecked;
00051   double norm;
00052   int posInGraphVector;
00053
00054  public:
00055   GraphVertex(Pointer data, Graph* owner);
00056   ~GraphVertex();
00057   void setPos(int pos) { this->posInGraphVector = pos; };
00058   int getPos() { return this->posInGraphVector; };
00059   void addNeighbor(Pointer neighbor) { this->neighbors->insert(neighbor); };
00060   void addReverse(Pointer reverse) { this->reverse->insert(reverse); };
00061   void removeNeighbor(Pointer neighbor, CompareFunc compare) {
00062     this->neighbors->remove(neighbor, compare);
00063   };
00064   void removeReverse(Pointer reverse, CompareFunc compare) {
00065     this->reverse->remove(reverse, compare);
00066   };
00067   PQueue* getNeighbors() { return this->neighbors; };
00068   PQueue* getReverse() { return this->reverse; };
00069   Pointer getData() { return this->data; };
00070   Graph* getOwner() { return this->owner; };
00071   void check() { this->hasBeenChecked = true; };
00072   bool checked() { return this->hasBeenChecked; };
00073   void setNorm(double norm) { this->norm = norm; };
00074   double getNorm() { return this->norm; };
00075 };
00076
00077 class GraphVertexPair {
00078  public:
00079   GraphVertexPair(Graph* owner, Pointer vertex1, Pointer vertex2)
00080       : flag(true), vertex1(vertex1), vertex2(vertex2), owner(owner){};
00081   Pointer getVertex1() { return this->vertex1; };
00082   Pointer getVertex2() { return this->vertex2; };
00083   Graph* getOwner() { return this->owner; };
00084   void setFalse() { this->flag = false; };
00085   bool getFlag() { return this->flag; };
00086
00087  private:
00088   bool flag;
00089   Pointer vertex1;
00090   Pointer vertex2;
00091   Graph* owner;
00092 };
```

7.3 ADTPQueue.hpp File Reference

`#include ¨ADTVector.hpp¨`
Include dependency graph for ADTPQueue.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class PQueue

  ADT Priority Queue.

7.3.1 Detailed Description

Author

Konstantinos Chousos

Version

0.1

Date

2023-10-30

Copyright

7.4 ADTPQueue.hpp

Go to the documentation of this file.

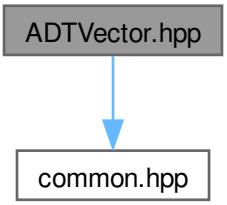
```
00001
00011 #pragma once
00012
00013 #include "ADTVector.hpp"
00014
00019 class PQueue {
00020  public:
00036   PQueue(CompareFunc compare, DestroyFunc destroyValue, Vector* values);
00041   ~PQueue();
00047   int getSize();
00053   Pointer getMax();
00059   Pointer getMin();
00065   void insert(Pointer value);
00070   void removeMax();
00077   DestroyFunc setDestroyValue(DestroyFunc destroyValue);
00086   void remove(Pointer value, CompareFunc compare);
00094   int find(Pointer value, CompareFunc compare);
00095
00101   Vector* toVector() { return this->vector; };
00102
00103   // Helper functions
00104   // These are used because the node IDs are 1-based, where as the
00105   // vector is 0-based.
00106   Pointer nodeValue(int nodeId);
00107   void nodeSwap(int nodeId1, int nodeId2);
00108   void bubbleUp(int nodeId);
00109   void bubbleDown(int nodeId);
00110   void naiveHeapify(Vector* values);
00111
00112  private:
00113   Vector* vector;
00114   CompareFunc compare;
00115   DestroyFunc destroyValue;
00116 };
```

7.5 ADTVector.hpp File Reference

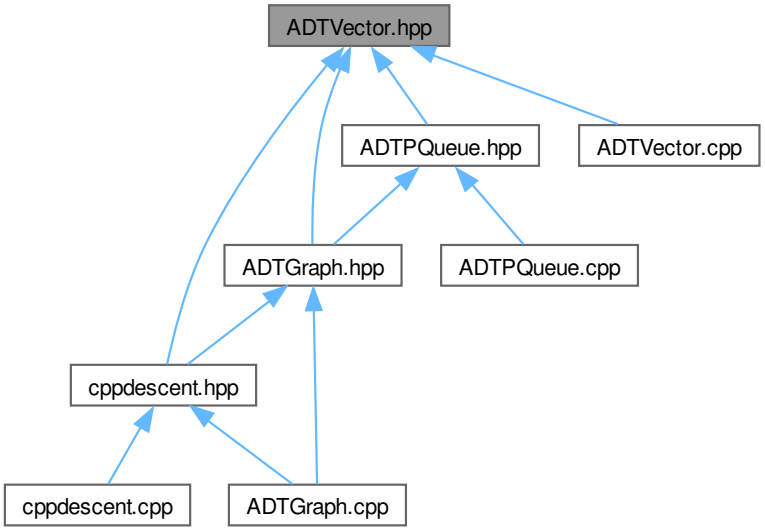
ADTVector implementation using a dynamic array.

```
#include ¨common.hpp¨
```

Include dependency graph for ADTVector.hpp:



This graph shows which files directly or indirectly include this file:



Classes

- class vectorNode

  Class for the vector Node object.
- class Vector

  ADT Vector.

Macros

- #define **VECTOR_MIN_CAPACITY** 10
- #define **VECTOR_BOF** (vectorNode∗)0
- #define **VECTOR_EOF** (vectorNode∗)0

7.5.1 Detailed Description

ADTVector implementation using a dynamic array.

Author

Konstantinos Chousos

Version

0.1

Date

2023-10-29

Copyright

7.6 ADTVector.hpp

Go to the documentation of this file.

```
00001
00011 #pragma once
00012
00013 #include "common.hpp"
00014
00015 #define VECTOR_MIN_CAPACITY 10
00016
00017 #define VECTOR_BOF (vectorNode*)0
00018 #define VECTOR_EOF (vectorNode*)0
00019
00026 class vectorNode {
00027  public:
00033   vectorNode(){};
00034   vectorNode(Pointer value) : value(value){};
00040   Pointer getValue() const { return value; };
00046   void setValue(Pointer value) { this->value = value; };
00047
00048  private:
00049   Pointer value;
00050 };
00051
00058 class Vector {
00059  public:
00073   Vector(int size, DestroyFunc destroyValue);
00080   ~Vector();
00086   int getSize();
00092   void insertLast(Pointer value);
00098   int removeLast();
00105   Pointer getAt(int pos);
00113   int setAt(int pos, Pointer value);
00121   Pointer find(Pointer value, CompareFunc compare);
00129   Pointer binaryFind(Pointer value, CompareFunc compare);
00137   int findPos(Pointer value, CompareFunc compare);
00144   DestroyFunc setDestroyValue(DestroyFunc destroyValue);
00150   vectorNode* first();
00156   vectorNode* last();
00163   vectorNode* next(vectorNode* node);
00170   vectorNode* previous(vectorNode* node);
00177   Pointer nodeValue(vectorNode* node);
00185   vectorNode* findNode(Pointer value, CompareFunc compare);
00193   void swap(int pos1, int pos2);
00194
00195  private:
00196   vectorNode* array;
00197   int size;
00198   int capacity;
00199   DestroyFunc destroyValue;
00200 };
```

7.7 common.hpp

```
00001 #pragma once
00002
00003 typedef void* Pointer;
00004
00005 typedef int (*CompareFunc)(Pointer a, Pointer b);
00006
00007 typedef void (*DestroyFunc)(Pointer value);
00008
00009 typedef unsigned int (*HashFunc)(Pointer);
```

7.8 cppdescent.hpp

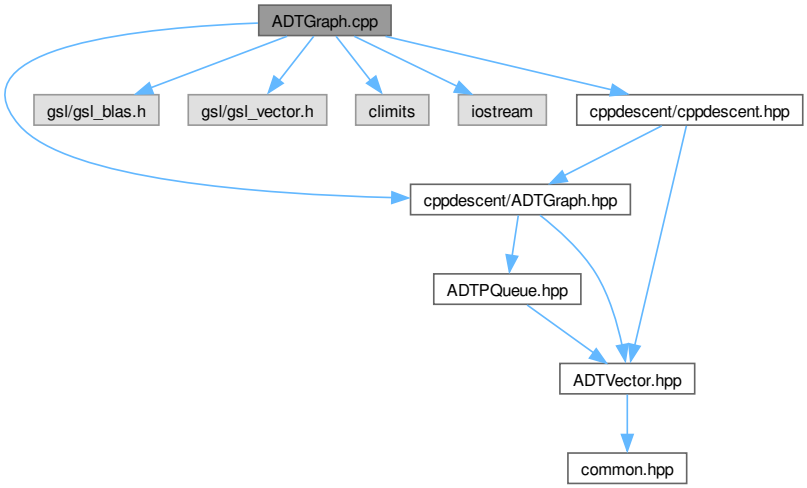
```
00001
00005 #include "ADTGraph.hpp"
00006 #include "ADTVector.hpp"
00007
00008 typedef float (*DistanceFunc)(Pointer a, Pointer b);
00009 extern bool verbose;
00010
00015 namespace cppdescent {
00016 int compareGraphVertices(Pointer vertex1, Pointer vertex2);
00017
00018 int compareGraphVertexPairs(Pointer p1, Pointer p2);
00019
00020 // ================================ I/O ====================================
00035 Vector* readBinData(const char* fp, int dimensions);
00036
00053 void writeBinGraph(const char* fp, Graph* graph, int K);
00054
00063 Graph* readBinGraph(const char* fp, int dimensions);
00064
00065 // ============================ Helper Functions ============================
00076 float recall(Graph* bfGraph, Graph* nnGraph, int N, int K);
00077
00078 // ============================ Metric Functions ============================
00087 float euclideanDistance(Pointer a, Pointer b);
00096 float manhattanDistance(Pointer a, Pointer b);
00104 int compareEdgesEuclidean(Pointer first, Pointer second);
00105
00106 int compareVertexPairsEuclidean(Pointer first, Pointer second);
00114 int compareEdgesManhattan(Pointer first, Pointer second);
00115
00116 // ============================ KNN computation ============================
00130 Graph* KNNBruteForceGraph(Vector* data, int K, CompareFunc compare);
00149 Graph* NNDescent_KNNGraph(Vector* data,
00150                           int K,
00151                           int D,
00152                           int Trees,
00153                           float delta,
00154                           float rho,
00155                           DistanceFunc distance);
00169 PQueue* NNDescent_Query(Graph* graph,
00170                         int K,
00171                         CompareFunc compare,
00172                         Vector* query);
00173
00174 // ========================= Random Projection Trees =========================
00175
00191 void RPTree(Graph* graph, Vector* vec, int K, int D, int dimensions);
00192
00193 };  // namespace cppdescent
```

7.9 ADTGraph.cpp File Reference

```
#include ¨cppdescent/ADTGraph.hpp¨
#include <gsl/gsl_blas.h>
#include <gsl/gsl_vector.h>
#include <climits>
#include <iostream>
```

```
#include ¨cppdescent/cppdescent.hpp¨
```

Include dependency graph for ADTGraph.cpp:



Functions

- int **compareVertices** (Pointer vertex1, Pointer vertex2)
- int **compareNeighbors** (Pointer neighbor1, Pointer neighbor2)
- void **destroyVertex** (GraphVertex ∗vertex)
- void **destroyEdgePair** (GraphVertexPair ∗pair)

7.9.1 Detailed Description

Author

Pheadon Seitanidis

Version

0.1

Date

2023-11-01

Copyright

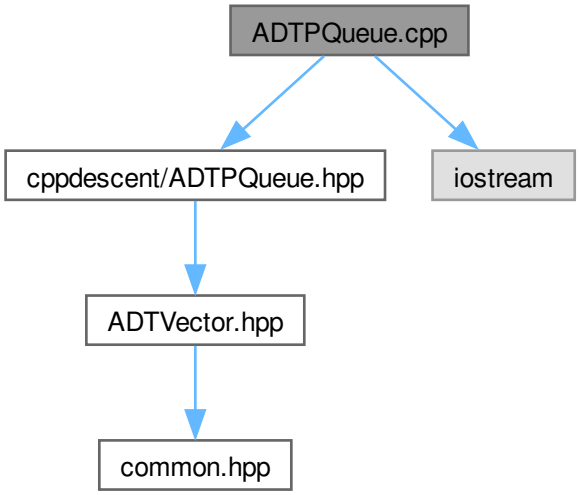
Copyright (c) 2023

7.10 ADTPQueue.cpp File Reference

An ADT Priority Queue implemented using a heap.

```
#include ¨cppdescent/ADTPQueue.hpp¨
#include <iostream>
```

Include dependency graph for ADTPQueue.cpp:



7.10.1 Detailed Description

An ADT Priority Queue implemented using a heap.

Author

Konstantinos Chousos

Version

0.1

Date

2023-10-30

Copyright

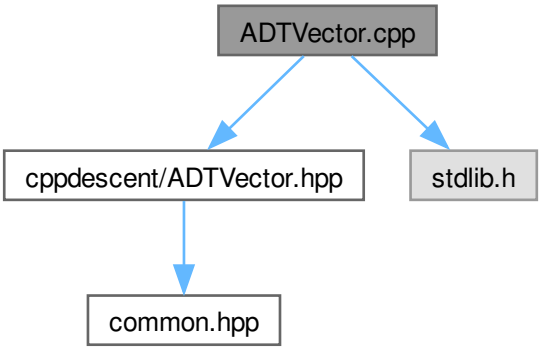
Copyright (c) 2023

7.11 ADTVector.cpp File Reference

Implementation of ADTVector using a dynamic array.

```
#include ¨cppdescent/ADTVector.hpp¨
#include ¨stdlib.h¨
```

Include dependency graph for ADTVector.cpp:



7.11.1 Detailed Description

Implementation of ADTVector using a dynamic array.

Author

Konstantinos Chousos

Version

0.1

Date

2023-10-30

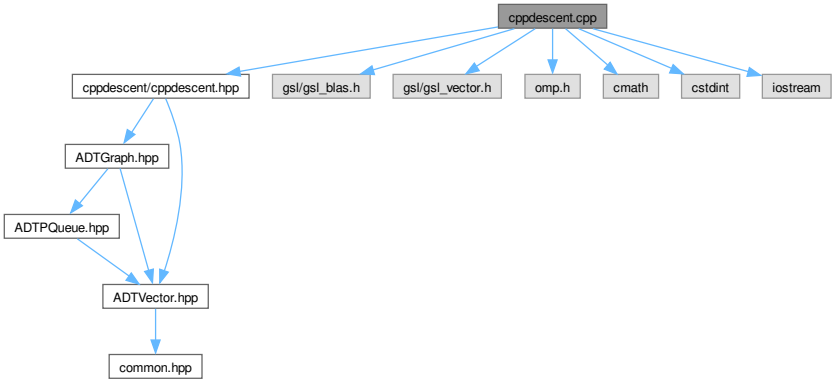
Copyright

7.12 cppdescent.cpp File Reference

Implementation of the cppdescent library.

```
#include ¨cppdescent/cppdescent.hpp¨
#include <gsl/gsl_blas.h>
#include <gsl/gsl_vector.h>
#include <omp.h>
#include <cmath>
#include <cstdint>
#include <iostream>
```

Include dependency graph for cppdescent.cpp:



Classes

- struct sets

Functions

- void **destroyEdges** (GraphVertexPair ∗pair)
- Graph ∗ sampleGraph (Vector ∗data, int K)

  Creates a random graph, where each vertex has K random neighbors.
- int **updateNN** (Graph ∗graph, int K, Pointer u1, Pointer u2, float dist, DistanceFunc distance)
- struct sets getSets (Vector ∗neighbors, int K, float rho)

  Get the Sets object.

Variables

- bool **verbose** = true

7.12.1 Detailed Description

Implementation of the cppdescent library.

Author

Konstantinos Chousos

Version

0.1

Date

2023-10-27

Copyright

Copyright (c) 2023

7.12.2 Function Documentation

7.12.2.1 getSets()

```
struct sets getSets (
            Vector * neighbors,
            int K,
            float rho )
```

Get the Sets object.

Returns a `sets` struct containing a vector pointer to the new[v] set and another to the old[v] set.

The first contains rho∗K of direct neighbors with their flag equal to true and rho∗K reverse neighbors with true. In other words, it contains 2∗rho∗K neighbors with flag = true.

The second contains all of the direct neighbors with flag = false, which in the worst case will be K, and rho∗K of the reverse neighbors with flag = false. In other words, K + rho∗K neighbors with flag = false.

Parameters

neighbors	
K	
rho	

Returns

struct sets

7.12.2.2 sampleGraph()

```
Graph * sampleGraph (
            Vector * data,
            int K )
```

Creates a random graph, where each vertex has K random neighbors.

The user is responsible for deallocating the graph.

Parameters

data	The Vector with all the points.
K	
compare	
distance	

Returns

Graph* The created graph.