cs280s21-b.sg

Dashboard / My courses / cs280s21-b.sg / General / Assignment 4: Graphs

Description Submission Edit Submission view

Grade

Reviewed on Thursday, April 1, 2021, 6:40 PM by Automatic grade
grade: 91.67 / 100.00

Assessment report[-]
[+]**Test 12: TestBig(99, 2)**
[+]**Summary of tests**

Submitted on Thursday, April 1, 2021, 6:39 PM (Download)

ALGraph.h

```
/******************************************************************************/
/*!
\file:      ALGraph.h
\author:    Goh Wei Zhe, weizhe.goh, 440000119
\par email: weizhe.goh\@digipen.edu
\date:      March 28, 2021
\brief      This file contains the declarations needed to construct a graph and
            implementing Dijkstra's algorithm.

Copyright (C) 2021 DigiPen Institute of Technology.
Reproduction or disclosure of this file or its contents without the
prior written consent of DigiPen Institute of Technology is prohibited.
*/
/******************************************************************************/

//---------------------------------------------------------------------------
#ifndef ALGRAPH_H
#define ALGRAPH_H
//---------------------------------------------------------------------------
#include <vector>
#include <algorithm>
#include <queue>
#include <list>
#include <iostream>

struct DijkstraInfo
{
  unsigned cost;
  std::vector<unsigned> path;
};

struct AdjacencyInfo
{
  unsigned id;
  unsigned weight;
};

typedef std::vector<std::vector<AdjacencyInfo> > ALIST;

class ALGraph
{
  public:
    ALGraph(unsigned size);
    ~ALGraph(void);
    void AddDEdge(unsigned source, unsigned destination, unsigned weight);
    void AddUEdge(unsigned node1, unsigned node2, unsigned weight);

    std::vector<DijkstraInfo> Dijkstra(unsigned start_node) const;
    ALIST GetAList(void) const;

  private:

    // An EXAMPLE of some other classes you may want to create and
    // implement in ALGraph.cpp
    class GNode;
    class GEdge;

    struct AdjInfo
    {
     // GNode *node;
      unsigned id;
      unsigned weight;

      //unsigned cost;
      //AdjInfo();
      bool operator<(const AdjInfo& rhs) const;
      bool operator>(const AdjInfo& rhs) const;
    };

    // Other private fields and methods
    struct NodeInfo
    {
        unsigned id;
        unsigned cost;
        unsigned prev;

        bool operator < (const NodeInfo& rhs) const;
        bool operator == (const unsigned& value) const;
    };

    unsigned size_;
    std::vector<std::vector<AdjInfo>> graph;

    unsigned extract_min(std::vector<AdjacencyInfo>& pq) const;
};

#endif

```

ALGraph.cpp

```
/******************************************************************************/
/*!
\file:      ALGraph.cpp
\author:    Goh Wei Zhe, weizhe.goh, 440000119
\par email: weizhe.goh\@digipen.edu
\date:      March 28, 2021
\brief      This file contains the definitions needed to construct a graph and
            implementing Dijkstra's algorithm.

Copyright (C) 2021 DigiPen Institute of Technology.
Reproduction or disclosure of this file or its contents without the
prior written consent of DigiPen Institute of Technology is prohibited.
*/
/******************************************************************************/
#include "ALGraph.h"

const unsigned INFINITY_ = static_cast<unsigned>(-1);

/******************************************************************************/
/*!
\fn     ALGraph::ALGraph(unsigned size)

\brief  Constructor for ALGraph
*/
/******************************************************************************/
ALGraph::ALGraph(unsigned size):size_{size}
{
    //fill the vector of graph with vector of AdjInfo
    for(unsigned i = 0; i < size; ++i)
    {
        graph.push_back(std::vector<AdjInfo>());
    }
}

/******************************************************************************/
/*!
\fn     ALGraph::~ALGraph(void)

\brief  Destructor for ALGraph
*/
/******************************************************************************/
ALGraph::~ALGraph(void){}

/******************************************************************************/
/*!
\fn     void ALGraph::AddDEdge(unsigned source, unsigned destination,
        unsigned weight)

\brief  Adds directed edge between two nodes.

\param  source - The starting node

\param  destination - The destination node

\param  weight - Cost to get from source to destination

*/
/******************************************************************************/
void ALGraph::AddDEdge(unsigned source, unsigned destination, unsigned weight)
{
    auto i = source - 1;
    graph[i].push_back(AdjInfo{destination, weight});

    auto sortLambda = [](AdjInfo& LHS, AdjInfo& RHS)
    {
        return (LHS.weight == RHS.weight)
        ? (LHS.id < RHS.id) : (LHS.weight < RHS.weight);
    };

    std::sort(graph[i].begin(), graph[i].end(), sortLambda);
}

/******************************************************************************/
/*!
\fn     void ALGraph::AddUEdge(unsigned node1, unsigned node2, unsigned weight)

\brief  Adds undirected nodes between two nodes

\param  node1 - first node

\param  node2 - second node

\param  weight - cost to get from node1 to node 2 and node2 to node 1

*/
/******************************************************************************/
void ALGraph::AddUEdge(unsigned node1, unsigned node2, unsigned weight)
{
    AddDEdge(node1, node2, weight);
    AddDEdge(node2, node1, weight);
}

/******************************************************************************/
/*!
\fn     std::vector<DijkstraInfo> ALGraph::Dijkstra(unsigned start_node) const

\brief  Performs Dijkstra's algorithm on the graph to find shortest path from a
        starting node to every possible node.

\param  start_node - the node that we are starting at.

\return Returns the result of Dijkstra algorithm that constaints a vector of
        DjkstraInfo
*/
/******************************************************************************/
std::vector<DijkstraInfo> ALGraph::Dijkstra(unsigned start_node) const
{
    std::vector<AdjacencyInfo> pq;
    std::vector<DijkstraInfo> dijkstra;

    for(unsigned i = 0; i < size_ ; ++i)
    {
        DijkstraInfo di;

        if(i + 1 == start_node)
        {
         di.cost = 0;
         di.path.push_back(start_node);
        }
        else
         di.cost = INFINITY_;

        dijkstra.push_back(di);
        pq.push_back(AdjacencyInfo{i+1, di.cost});
    }

    while(!pq.empty())
    {
        //Returns index of min cost node
        unsigned u = extract_min(pq);

        //if extracted cost is infinite, it is not updated and is not connected
        //to any nodes

        if(dijkstra[u-1].cost == INFINITY_)
            break;

        //for each neighbour v of u
        for(unsigned v = 0; v < graph[u - 1].size(); ++v)
        {
            //if dist[u] + cost(u,v) < dist[v]
            if(dijkstra[u - 1].cost + graph[u-1][v].weight <
            dijkstra[graph[u-1][v].id-1].cost)
            {
                //update cost
                dijkstra[graph[u-1][v].id-1].cost =
                dijkstra[u-1].cost + graph[u-1][v].weight;

                //update previous path
                dijkstra[graph[u-1][v].id-1].path = dijkstra[u-1].path;
                dijkstra[graph[u-1][v].id-1].path.push_back(graph[u-1][v].id);

                //Update info in priority_queue
                for(unsigned i = 0; i < pq.size(); ++i)
                {
                    if(pq[i].id == graph[u-1][v].id)
                    {
                        //update cost
                        pq[i].weight = dijkstra[graph[u-1][v].id-1].cost;
                    }
                }
            }
        }

        //Remove minimum cost node from priority queue
        for(unsigned i = 0; i < pq.size(); ++i)
        {
            if(pq[i].id == u)
                pq.erase(pq.begin() + i);
        }
    }

    return dijkstra;
}

/******************************************************************************/
/*!
\fn     ALIST ALGraph::GetAList(void) const

\brief  Gets adjacency matrix for the graph

\return Returns adjacency matrix of the graph
*/
/******************************************************************************/
ALIST ALGraph::GetAList(void) const
{
    ALIST adjList;

    //copy graph to adjList matrix
    for(auto i : graph)
    {
        std::vector<AdjacencyInfo> v;

        for(auto j : i)
        {
            v.push_back(AdjacencyInfo{j.id, j.weight});
        }

        adjList.push_back(v);
    }
    return adjList;
}

/******************************************************************************/
/*!
\fn     bool ALGraph::AdjInfo::operator < (const AdjInfo& rhs) const

\brief  Compares to AdjacencyInfo node for sorting

\param  rhs - right hand side node

\return Returns true if weight is less than rhs.weight, else, return false.
*/
/******************************************************************************/
```

```
bool ALGraph::AdjInfo::operator < (const AdjInfo& rhs) const
{
    return weight < rhs.weight;
}

/******************************************************************************/
/*!
\fn     bool ALGraph::AdjInfo::operator > (const AdjInfo& rhs) const

\brief  Compares to AdjacencyInfo node for sorting

\param  rhs - right hand side node

\return Returns true if weight is more than rhs.wight, else, return false.
*/
/******************************************************************************/
bool ALGraph::AdjInfo::operator > (const AdjInfo& rhs) const
{
    return weight > rhs.weight;
}

/******************************************************************************/
/*!
\fn     unsigned ALGraph::extract_min(std::vector<AdjacencyInfo>& pq) const

\brief  Extract the node with the minimum cost

\param  pq - priorty queue that contains a vector of nodes

\return Returns the node index with the minimum cost
*/
/******************************************************************************/
unsigned ALGraph::extract_min(std::vector<AdjacencyInfo>& pq) const
{
    if(pq.empty())
        return 0;

    unsigned id = pq[0].id;
    unsigned min_weight = INFINITY_;

    for(unsigned i = 0; i < pq.size(); ++i)
    {
        if(pq[i].weight < min_weight)
        {
            id = pq[i].id;
            min_weight = pq[i].weight;
        }
    }

    return id;
```

VPL

◄ Assignment 3: AVL Trees

Jump to... ⬍

Assignment 5: Hashing ►

You are logged in as Wei Zhe GOH (Log out)
cs280s21-b.sg
Data retention summary
Get the mobile app