

**计算机科学与工程学院**

**《探究式学习训练3》实验报告**

项目名称：基于Dijkstra算法的路由器PRIM路由协议设计与实现

专 业：计算机科学与技术

班 级：计算机232班

姓 名：梁家祥

完成日期：2024.12.30

目录

[1. 问题描述 3](#_Toc155091992)

[2. 需求分析 3](#_Toc155091993)

[3. 概要设计 4](#_Toc155091994)

[4. 详细设计 4](#_Toc155091995)

[5. 编码与调试分析 5](#_Toc155091996)

[6. 运行成果 5](#_Toc155091997)

[7. 总结 5](#_Toc155091998)

# 问题描述

设计一个基于Dijkstra算法的路由器PRIM路由协议系统，该系统旨在模拟和解决计算机网络中的路由选择问题，确保数据包能够高效地从源节点传输到目标节点。

基本要求：

1.存储结构：使用合适的数据结构（如图、邻接矩阵或邻接表）存储路由器节点及其之间的连接关系（包括连接权重，代表传输成本或延迟）。

2.Dijkstra算法实现：实现Dijkstra算法，用于计算从源节点到所有其他节点的最短路径。算法应考虑连接权重，确保路径选择的优化。

3.PRIM路由协议：基于Dijkstra算法的结果，设计并实现PRIM路由协议，该协议应能够动态地更新路由表，反映网络拓扑的变化，并根据最短路径选择最优路由。

4.用户界面与交互：设计一个用户友好的界面，允许用户输入网络拓扑信息（节点、连接及权重），选择源节点，并展示Dijkstra算法计算出的最短路径及PRIM路由协议生成的路由表。

5.扩展功能：可以添加网络拓扑的动态更新功能，模拟网络中的节点加入、离开或连接状态的变化，并实时更新路由表。同时，可以引入多线程或异步处理，提高系统的响应速度和并发处理能力。

6.性能评估：实现一个性能评估模块，用于测试系统在不同网络规模、连接密度和权重分布下的运行效率和准确性。

# 需求分析

* **存储结构**
  + 使用合适的数据结构（如图、邻接矩阵或邻接表）存储路由器节点及其之间的连接关系。
  + 连接关系包括连接权重，代表传输成本或延迟。
* **Dijkstra算法实现**
  + 实现Dijkstra算法，用于计算从源节点到所有其他节点的最短路径。
  + 算法应考虑连接权重，确保路径选择的优化。
* **路由协议**
  + 基于Dijkstra算法的结果，设计并实现路由协议。
  + 该协议应能够动态地更新路由表，反映网络拓扑的变化，并根据最短路径选择最优路由。
* **用户界面与交互**
  + 设计一个用户友好的界面，允许用户输入网络拓扑信息（节点、连接及权重）。
  + 用户可以选择源节点，并展示Dijkstra算法计算出的最短路径及路由协议生成的路由表。
* **扩展功能**
  + 添加网络拓扑的动态更新功能，模拟网络中的节点加入、离开或连接状态的变化，并实时更新路由表。
  + 引入多线程或异步处理，提高系统的响应速度和并发处理能力。
* **性能评估**
  + 实现一个性能评估模块，用于测试系统在不同网络规模、连接密度和权重分布下的运行效率和准确性。

# 概要设计

* 储存结构
  + 使用邻接表存储路由器结点极其之间的连接关系；
  + 为了方便路由的查找和删除，使用两个哈希表构造邻接表。第一层哈希表的键代表出发顶点，第二层哈希表的键值对代表到达顶点和连接权重。
* Dijkstra算法实现
  + 使用优先队列法实现开销较低的Dijkstra算法。
* 路由协议
  + 模拟OSPF（Open Shortest Path First）协议，以每个路由器为一个原点，通过Dijkstra算法生成最短路径；
  + 网络拓扑结构由一个全局网关类NetworkManager进行存储。
* 用户界面与交互
  + 使用Gtk进行图形化界面的开发；
  + 用户可以编辑网络拓扑信息，并在DrawingArea内自由放置、拖动节点。
* 性能评估
  + 使用Benchmark套件测试项目实现的Dijkstra算法在不同网络拓扑情况下的性能开销；
  + 基于CTest，使用Boost库的Dijkstra算法检验项目实现的Dijkstra算法的正确性。

# 详细设计

* 存储结构
  + 哈希表的实现：使用键值对的数据结构作为哈希表的Bucket，使用C++内置数组进行线性空间的分配；

1. template<typename Key, typename Value>

2. struct Bucket {

3. Bucket(const Key &key, const Value &value): key(key), value(value) {

4. }

5. Key key;

6. Value value;

7. };

8. // 哈希表构造函数

9. template<typename Key, typename Value>

10. Hashmap<Key, Value>::Hashmap(std::function<uint32\_t(Key)> hashFunction): hashCompute(hashFunction) {

11. Hashtable = new Bucket<Key, Value> \*[size];

12. for (uint32\_t i = 0; i < size; i++) {

13. Hashtable[i] = nullptr;

14. }

15. }

该哈希表采用线性探测法检测并避免哈希冲突。

1. bool has\_key = false;

2. while (start != end) {

3. if (Hashtable[start] == nullptr) break;

4. if (Hashtable[start]->key == key) {

5. has\_key = true;

6. break;

7. }

8. start = (start + 1) % size;

9. }

* + 邻接表的实现：使用双层嵌套的哈希表构造邻接表。

1. template<typename T, typename WEIGHT\_T>

2. class Graph {

3. public:

4. //……

5. private:

6. ConcurrentHashmap<T, ConcurrentHashmap<T, WEIGHT\_T> \*> adjList = ConcurrentHashmap<

7. T, ConcurrentHashmap<T, WEIGHT\_T> \*>();

8. };

* Dijkstra算法实现

为了在每个路由节点上隔离运行Dijkstra算法，我们使用组合模式分离出DijkstraResolver类，在其resolve方法中实现Dijkstra算法。

1. class DijkstraResolver {

2. //…

3. private:  
 T origin;  
 Hashmap<T, T> predecessors;  
 Hashmap<T, WEIGHT\_T> dis;

4. }

1. void DijkstraResolver<T, WEIGHT\_T>::resolve(Graph<T, WEIGHT\_T> \*graph) {

2. PriorityQueue<T> queue;

3. Hashmap<T, bool> visited;

4.

5. graph->visitAllNode([this,&visited](T &u) {

6. dis.set(u, MaxValue<WEIGHT\_T>::value);

7. visited.set(u, false);

8. });

9. dis.set(origin, 0);

10. queue.enqueue(origin, 0);

11. while (!queue.isEmpty()) {

12. T u = queue.pop();

13. if (visited.get(u)) continue;

14. visited.set(u, true);

15. graph->visitAllEdge(u, [this,&u,&queue](T v, WEIGHT\_T w) {

16. if (dis.get(v) > dis.get(u) + w) {

17. dis.set(v, dis.get(u) + w);

18. queue.enqueue(v, dis.get(v));

19. predecessors.set(v, u);

20. }

21. });

22. }

23. }

该Dijkstra算法会初始化在给定的图中遍历所有节点，先初始化所有节点的距离

# 编码与调试分析

# 运行成果

# 总结