

**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

[一、 问题描述 3](#_Toc186470208)

[二、 需求分析 3](#_Toc186470209)

[三、 概要设计 3](#_Toc186470210)

[四、 详细设计 4](#_Toc186470211)

[五、 编码与调试分析 8](#_Toc186470212)

[六、 运行成果 9](#_Toc186470213)

[七、 总结 10](#_Toc186470214)

# 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算法会初始化访问标记哈希表visited、节点对应距离哈希表dis，随后在给定的图中遍历所有节点，对节点的每一条边进行松弛操作。边上的节点会随着边权入队最小优先队列，以保证每一次遍历出队的节点的边权最小。

* **路由协议**
  + 定义路由节点类Router。该类包含了路由的基本信息——路由名称、路由延迟、路由UUID及其哈希。其中，路由延迟用来确定网络拓扑结构中两条边的边权，路由UUID及其生成的哈希用来确认路由在图结构哈希表中的位置，减少哈希冲突发生的可能性。

1. class Router {

2. public:

3. //…

4. private:

5. std::string \_uuid;

6. std::string \_name;

7. uint16\_t \_delay{};

8. uint32\_t \_hash;

9. }

* + 定义全局网关类NetworkManager，使用单例模式来确保该类的全局唯一性。该类带有Graph类成员，用于储存网络拓扑结构；该类还有一个定时器函数，在构造函数中启动，在析构函数中终止，用于每隔一段时间遍历图中所有节点并将Dijkstra任务推送到线程池中。

1. class NetworkManager {

2. public:

3. NetworkManager(const NetworkManager &obj) = delete;

4. NetworkManager &operator=(const NetworkManager &obj) = delete;

5.

6. ~NetworkManager();

7. static NetworkManager \*getInstance();

8. void addRouter(Router \*router);

9. void removeRouter(Router \*router);

10. uint32\_t getLineDelay(Router \*router\_s, Router \*router\_v);

11. void connect(Router \*router1, Router \*router2);

12. void connect(Router \*router1, Router \*router2, uint16\_t weight);

13. Graph<Router \*, uint16\_t> \*get\_graph() { return &graph; }

14. void stopTimer();

15.

16. private:

17. NetworkManager();

18. void resolveTaskTimer();

19. static NetworkManager \*\_instance;

20. Graph<Router \*, uint16\_t> graph;

21. std::atomic<bool> timer\_stop = false;

22. std::thread resolve\_thread;

23. BS::thread\_pool<> pool;

24. }

* **用户界面与交互**
  + 使用Gtkmm进行图形界面的开发。由于需要在Drawing Area中储存节点的坐标位置，我们新建一个基于RouterNode结构体的Graph模板类来管理Drawing Area中的节点。

1. struct RouterNode {

2. RouterNode(const std::shared\_ptr<Router> &r, const double xPos, const double yPos,

3. const bool sel) : router(r), x(xPos), y(yPos), selected(sel) {

4. }

5.

6. std::shared\_ptr<Router> router;

7. double x, y;

8. bool selected;

9. bool onShortestPath = false;

10. };

11. Graph<std::shared\_ptr<RouterNode>, uint16\_t> router\_graph;

* + 使用继承自Gtk::Box的各种RouterAction类型来对Drawing Area中的节点进行操作。
  + 在RouterAction上方是RouterInfo类，用于显示单个节点的各种信息。
* **性能评估**
  + 使用Google Benchmark来测试Dijkstra算法在不同网络拓扑情况下每次运行时的性能开销。

1. static void BM\_RouterDijkstraResolve(benchmark::State &state) {

2. //引入随机库…

3. Router \*router\_array[state.range()];

4. auto nm = NetworkManager::getInstance();

5. nm->stopTimer();//停止多线程任务

6.

7. for (int i = 0; i < state.range(); i++) {

8. //添加足量路由节点

9. }

10. uint32\_t edges = 0;

11. while (edges < state.range()) {

12. //生成足量的边

13. }

14. Router \*another\_router\_ = router\_array[distro\_(gen\_)]; //随机一点反复进行Dijkstra算法

15. for (auto \_: state) {

16. another\_router\_->resolve();

17. }

18. for (const Router \*router: router\_array) {

19. delete router;

20. }

21. }

22.

23. BENCHMARK(BM\_RouterDijkstraResolve)->Arg(100);

24. BENCHMARK(BM\_RouterDijkstraResolve)->Arg(500);

25. BENCHMARK(BM\_RouterDijkstraResolve)->Arg(1000);//…

* 基于CTest，使用Boost库的Dijkstra算法来检测该项目Dijkstra算法的正确性。

1. void test(uint64\_t num\_nodes, uint64\_t num\_edges) {

2. //初始化数据结构

3. uint64\_t cnt\_edges = 0;

4. while (cnt\_edges < num\_edges) {

5. //添加足量边

6. }

7.

8. BoostGraph boost\_graph(edge\_array.begin(), edge\_array.end(), weights.begin(), num\_nodes);//Boost库图结构

9.

10. std::vector<Vertex> predecessors(num\_nodes);

11. std::vector<uint64\_t> distances(num\_nodes);

12. Vertex start\_vertex = 0;

13. boost::dijkstra\_shortest\_paths(boost\_graph, start\_vertex,

14. boost::predecessor\_map(boost::make\_iterator\_property\_map(

15. predecessors.begin(), boost::get(boost::vertex\_index, boost\_graph))).

16. distance\_map(boost::make\_iterator\_property\_map(

17. distances.begin(), boost::get(boost::vertex\_index, boost\_graph))));

18. //取单次1/4或有值点作为Dijkstra原点

19. uint64\_t target\_vertex = num\_nodes / 4;

20. while (distances[target\_vertex] == std::numeric\_limits<uint64\_t>::max()) {

21. ++target\_vertex;

22. }

23.

24. std::vector<uint64\_t> boost\_path;

25. for (uint64\_t v = target\_vertex; v != start\_vertex; v = predecessors[v]) {

26. //处理Boost库路径

27. }

28.

29. Graph<uint64\_t, uint64\_t> lr\_graph; //该项目图数据结构

30. for (uint64\_t i = 0; i < num\_nodes; ++i) {

31. lr\_graph.addNode(i);

32. }

33. for (uint64\_t i = 0; i < num\_edges; ++i) {

34. lr\_graph.addEdge(edge\_array[i].first, edge\_array[i].second, weights[i]);

35. }

36.

37. DijkstraResolver<uint64\_t, uint64\_t> lr\_dijkstra\_resolver(start\_vertex);

38. lr\_dijkstra\_resolver.resolve(&lr\_graph);

39. Stack<uint64\_t> lr\_stack = lr\_dijkstra\_resolver.getShortestPath(target\_vertex);

40.

41. std::vector<uint64\_t> lr\_path;

42. while (!lr\_stack.isEmpty()) {

43. lr\_path.push\_back(lr\_stack.pop());

44. }

45. assert(boost\_path.size() == lr\_path.size());

46. std::cout << "Length Correct" << std::endl;

47.

48. assert(distances[target\_vertex] == lr\_dijkstra\_resolver.getShortestWeight(target\_vertex));

49. std::cout << "Distance Correct" << std::endl;

50.

51. while (!lr\_path.empty()) {

52. assert(lr\_path.back() == boost\_path.back());

53. lr\_path.pop\_back();

54. boost\_path.pop\_back();

55. }

56. std::cout << "Path Correct" << std::endl;

57. std::cout << "Test Complete!" << std::endl;

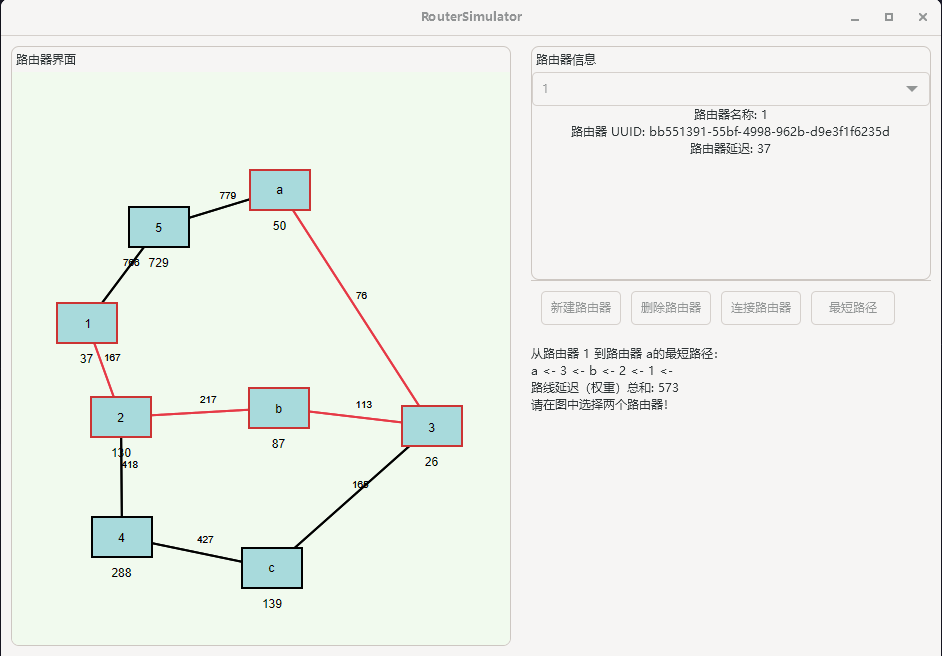
58. }

# 编码与调试分析

* 为了增强泛用型，本项目在哈希表、优先队列、栈等数据结构上使用了泛型并广泛运用；
* 本项目使用CMake构建，将图形化界面与路由节点代码分离为“RouterSimulator”和“librouter”两个项目，并在前者调用后者的过程中不断修改、完善接口；
* 使用GDB调试器可以调试项目代码，快速定位存在的问题。

# 运行成果

* **图形化界面截图展示：**



* **性能评估结果：**

1. Run on (16 X 4046.64 MHz CPU s)

2. CPU Caches:

3. L1 Data 32 KiB (x8)

4. L1 Instruction 32 KiB (x8)

5. L2 Unified 1024 KiB (x8)

6. L3 Unified 16384 KiB (x1)

7. Load Average: 0.81, 1.13, 1.12

8. -------------------------------------------------------------------------------

9. Benchmark Time CPU Iterations

10. -------------------------------------------------------------------------------

11. BM\_RouterDijkstraResolve/100 60212 ns 60171 ns 13973

12. BM\_RouterDijkstraResolve/500 1628534 ns 1627549 ns 430

13. BM\_RouterDijkstraResolve/1000 5924750 ns 5921240 ns 108

14. BM\_RouterDijkstraResolve/2000 45538957 ns 45512742 ns 18

15. BM\_RouterDijkstraResolve/5000 319310217 ns 319145715 ns 3

16. BM\_RouterDijkstraResolve/10000 1272111087 ns 1269666098 ns 1

17. BM\_RouterDijkstraResolve\_Boost/100 2422 ns 2418 ns 284388

18. BM\_RouterDijkstraResolve\_Boost/500 52201 ns 52105 ns 10000

19. BM\_RouterDijkstraResolve\_Boost/1000 346722 ns 346095 ns 2007

20. BM\_RouterDijkstraResolve\_Boost/2000 3584063 ns 3577963 ns 218

21. BM\_RouterDijkstraResolve\_Boost/5000 27324730 ns 27284660 ns 26

22. BM\_RouterDijkstraResolve\_Boost/10000 115234968 ns 115114017 ns 6

在性能评估中，面对大型网络节点运算时，该项目的Dijkstra实现能够在300ms内完成解算，实际运行在性能较差的路由器上时可能需要1~2s的时间。性能的劣势主要由哈希表遍历速度较慢造成。

正确性测试中，该项目通过了所有的随机测试。

# 总结

本项目成功设计了一个基于Dijkstra算法的路由协议系统，并实现了一系列为此提供支持的数据结构。在此基础上，引入多线程/异步方法，提高系统的整体流畅度；使用Gtk图形库开发用户界面，提供了友好的用户体验。在本次项目的开发过程中，我深入学习了多种数据结构、算法及其实现，并通过C++的模板、图形库和标准库进一步加深了我对数据结构的认知。