基于K-Means聚类的NS3 AODV优化路由算法的研究

成员姓名	成员分工
胡峻玮	K-means聚类算法的实现、AODV环境搭建、报告撰写
谌乐俊杰	K-means聚类算法的优化、AODV优化网络可视化、PPT汇报
杨英凡	AODV路由原始算法实现、AODV路由算法可视化、PPT制作
卢奕人	AODV路由原始算法实现、AODV路由算法可视化、PPT制作

一、摘要

1. 实验背景

- 在无线自组网（`Ad Hoc On-Demand Distance Vector, AODV`）路由协议中，路由发现过程是通过泛洪方法实现的。这种方法通过向发送方传输范围内的所有节点广播路由请求（`RREQ`）数据包来完成。然而，这种方式通常会导致大量不必要的`RREQ`数据包传输，并在响应中生成相应的路由回复（`RREP`）数据包。结果是数据包的重新传输增多，进而引发数据包冲突和网络拥塞问题。
- `AODV`路由协议因其简单高效而广泛应用于无线自组网络中，但其泛洪路由发现机制的局限性却是显而易见的。在高密度网络环境中，这种泛洪机制会导致过多的控制包传输，浪费网络资源，增加端到端延迟，甚至影响网络的整体性能。为了解决这一问题，优化`AODV`的路由发现方法成为一个重要的研究方向。
- 在这个项目中，我们提出了一种基于`K-Means`聚类算法的优化`AODV`路由发现方法。`K-Means`聚类算法是一种经典的无监督学习算法，能够有效地将数据点划分为若干簇。在我们的方案中，利用`K-Means`聚类算法选择最佳的RREQ数据包转发器集群，而不是简单的广播。这种方法的核心思想是通过聚类算法识别网络中的关键节点，集中这些节点进行RREQ数据包的转发，从而减少不必要的控制数据包传输。

2. 研究目的

- 本项目旨在通过优化`AODV`路由协议，减少无线自组网中的控制数据包传输量，降低网络拥塞和端到端延迟，从而提升网络的整体性能。具体目标包括：
  - 实现传统`AODV`路由协议。
  - 设计并实现基于`K-Means`聚类算法的`AODV`优化算法。
  - 在`NS3`仿真平台上验证优化算法的有效性。
  - 通过实验比较优化前后的性能差异，评估优化算法在高密度网络环境中的性能提升。

3. 主要方法

为实现上述目标，本项目采用以下主要方法：

- **原始AODV路由算法实现**：在NS3仿真平台上实现传统的AODV路由协议，作为对比基准。
- **K-Means聚类算法应用**：使用K-Means聚类算法对网络节点进行聚类，选取各个集群的中心节点作为RREQ数据包的主要转发器。

- **优化AODV路由算法实现**：在原始AODV路由协议的基础上，设计并实现基于K-Means聚类的优化算法，减少不必要的控制数据包传输。
- **实验验证**：通过在NS3仿真平台上的大量实验，比较原始AODV和优化AODV的性能，特别是在控制数据包传输量和端到端延迟方面的表现。

4. 关键词

`AODV`路由协议 `K-Means`聚类算法 网络优化 `NS3`仿真 控制数据包 端到端延迟

二、实验原理

1. 现有研究综述

- 无线自组网（`Ad Hoc Network`）是一种临时性的、无基础设施支持的网络结构，具有自组织和动态变化的特点。由于其灵活性和便捷性，广泛应用于军事通信、灾难救援、移动计算等领域。`AODV`（`Ad Hoc On-Demand Distance Vector`）路由协议是其中一种典型的按需路由协议，通过动态建立路由以应对网络的快速变化。
- 现有的研究主要集中在优化`AODV`的路由发现和路由维护机制，以提高网络的性能。例如，许多研究者提出了改进的路由发现方法，如限制泛洪范围、采用不同的路由选择标准等。此外，还有一些研究尝试引入机器学习和数据挖掘技术来优化路由协议，取得了显著的成果。

2. AODV路由协议介绍

- `AODV`路由协议是一种按需路由协议，其核心思想是只有当源节点需要发送数据包到目标节点时才建立路由。`AODV`采用一种简单的泛洪机制进行路由发现，并维护一个路由表来记录路由信息。其工作原理主要包括以下几个步骤：
  - **路由发现**：当源节点需要与目标节点通信时，首先会广播一个路由请求（`RREQ`）数据包，网络中的其他节点收到RREQ后，会检查自己是否是目标节点或是否有到目标节点的有效路由。如果是，则回复一个路由回复（`RREP`）数据包，否则继续转发`RREQ`。
  - **路由维护**：`AODV`通过定期发送`HELLO`消息和监测链路状态来维护路由表。如果发现链路断裂，会发送路由错误（`RERR`）消息通知相关节点，从而触发重新路由发现过程。
  - **路由删除**：当路由不再使用时，`AODV`会删除无效的路由条目，释放资源。

- 尽管`AODV`在动态网络环境中表现良好，但其泛洪机制带来的高控制开销和潜在的网络拥塞问题仍然需要解决。

3. K-Means聚类算法介绍

$K-means$是一种广泛使用的聚类算法，特别适合处理大数据集中的分群问题。它通过迭代过程将数据点划分到K个聚类中，以使每个点与其所属聚类的中心之间的距离之和最小。以下是该算法的详细介绍和公式推导：

- 算法步骤
  - **初始化**：随机选择K个数据点作为初始聚类中心。
  - **分配**：将每个数据点分配给最近的聚类中心。
  - **更新**：重新计算每个聚类的中心，即聚类中所有点的均值。

    - **迭代**：重复分配和更新步骤，直到聚类中心的变化小于某个阈值，或达到预设的最大迭代次数。
- 公式推导
    - 分配步骤
        - 在分配步骤中，计算每个数据点 (x) 到每个聚类中心 (c_i) 的欧氏距离。这个距离的计算公式为：

$$\text{dist}(x, c_i) = \sqrt{\sum_{j=1}^{d} (x_j - c_{ij})^2}$$

        其中，(x_j) 和 (c_{ij}) 分别是数据点 (x) 和聚类中心 (c_i) 在第 (j) 维的坐标值。
    - 更新步骤
        - 在更新步骤中，新的聚类中心是属于该聚类的所有点的均值：

$$c_i = \frac{1}{|S_i|} \sum_{x \in S_i} x$$

        这里 (S_i) 表示第 (i) 个聚类中的数据点集合，而 ($|S_i|$) 是该集合中数据点的数量。
- 终止条件
    - K-means算法通常继续迭代，直到聚类中心的变化非常小（小于一个预设的阈值），或者达到了最大迭代次数。这些终止条件确保算法在合理的时间内收敛并提供稳定的聚类结果。
- 注意事项
    - K-means算法对初始聚类中心的选择非常敏感，不恰当的选择可能导致次优的聚类结果。此外，算法结果也依赖于聚类数 (K) 的选择，不同的 (K) 值可能导致完全不同的聚类结果。为了获得更好的聚类效果，通常需要结合具体的应用背景来选择合适的 (K) 值，并可能需要多次运行算法以确保结果的稳定性。

4. 当前研究的不足与挑战

- 尽管现有研究在优化 `AODV` 路由协议方面取得了一些进展，但仍存在一些不足与挑战：
    - **控制开销高**：传统 `AODV` 协议采用泛洪机制进行路由发现，导致大量的控制数据包传输，增加了网络开销和拥塞。
    - **路由选择不优化**：现有的一些优化方法在选择路由时没有充分考虑网络的动态变化和节点的资源情况，可能导致次优路由选择。
    - **缺乏智能优化**：虽然引入了机器学习和数据挖掘技术，但应用效果尚未完全验证，特别是在实际网络环境中的应用效果有待进一步研究。
    - **算法复杂度和资源消耗**：一些改进方法增加了算法的复杂度和资源消耗，需要在性能提升和资源利用之间找到平衡。
- 本项目通过引入 `K-Means` 聚类算法来优化 `AODV` 路由发现过程，旨在减少控制数据包的传输量，降低网络拥塞，提高路由选择的智能性和整体网络性能。

三、环境配置

1. 环境依赖

本项目在`VMware`虚拟机下的`Linux Ubuntu 20.04`操作系统中进行开发，使用`NS-3.35`作为仿真平台。为了确保项目顺利进行，需要安装和配置以下环境依赖：

- 操作系统
  - **Ubuntu 20.04**：该操作系统稳定且广泛用于开发和研究。
- 仿真平台
  - **NS-3.35**：一个用于网络仿真的开源工具，具有强大的网络模拟功能。
- 编译器和工具
  - **GCC**：`GNU Compiler Collection`，用于编译`C++`代码。建议安装最新版本。
  - **GNU make**：一个自动化编译工具。
  - **Python**：用于运行`NS-3`的脚本和配置仿真环境，建议使用`Python 3`
- 其他依赖库
  - **GNU Scientific Library (GSL)**：用于科学计算的库。
  - **CMake**：一个跨平台的构建系统。
  - **Libxml2**：用于处理`XML`文件的库。

2. 环境配置

- **更新系统**

```
sudo apt-get update
sudo apt-get upgrade
```

- **安装基础工具**

```
sudo apt-get install build-essential autoconf automake libtool gcc g++ make
```

- **安装Python**

```
sudo apt-get install python3 python3-dev python3-pip
```

- **安装NS-3依赖**

```
sudo apt-get install gcc g++ python3 python3-dev python3-setuptools git mercurial qt5-qmake qt5-default gnuplot-x11 gir1.2-goocanvas-2.0 python3-gi python3-gi-cairo python3-pygraphviz python3-dev python3-pygoocanvas ipython3 openmpi-bin openmpi-common openmpi-doc libopenmpi-dev autoconf cvs bzr unrar gdb valgrind uncrustify doxygen graphviz imagemagick tcpdump sqlite sqlite3 libsqlite3-dev libgtk2.0-0 libgtk2.0-dev vtun lxc libxml2 libxml2-dev cmake libc6-dev libc6-dev-i386 libclang-dev llvm-dev automake
```

- **下载并安装NS-3.35**

```
wget https://www.nsnam.org/releases/ns-allinone-3.35.tar.bz2
tar -xjf ns-allinone-3.35.tar.bz2
cd ns-allinone-3.35/ns-3.35
./waf configure --build-profile=debug --enable-examples --enable-tests
./waf build
```

- **配置环境变量**
  将以下内容添加到`~/.bashrc`文件中：

```
export PYTHONPATH=$PYTHONPATH:/path/to/ns-allinone-3.35/ns-3.35/build/lib
export LD_LIBRARY_PATH=$LD_LIBRARY_PATH:/path/to/ns-allinone-3.35/ns-3.35/build/lib
```

- **运行**：

```
source ~/.bashrc
```

- 验证安装
  - 安装完成后，可以通过运行以下命令来验证NS-3是否安装成功：

```
cd ~/ns-allinone-3.35/ns-3.35
./waf --run hello-simulator
```

  - 如果看到类似以下的输出，则说明NS-3安装成功：

```
Hello Simulator
```

四、实验步骤

1. 构建实验环境

网络设置

- 网络规模

  节点数（`m_nodes`）：
  仿真网络中包含10-50个节点（节点数量为自变量）
  仿真网络中包含15个节点（pps为自变量）

- 节点移动性
  移动速度（`m_speed`）：节点的移动速度设定为20米/秒，均匀随机分布。
  暂停时间（`m_pause`）：节点的暂停时间为0秒。
  位置分布：节点初始位置在300x300米的区域内均匀随机分布。
  移动模型：节点采用随机游走模型，其速度在0到`m_speed`之间随机分布，暂停时间为`m_pause`秒。

- 通信技术
  无线通信标准：采用`IEEE 802.11b`标准进行无线通信。

- 路由协议
  AODV：网络使用AODV作为路由协议。

2. 创建实验项目

- 该步骤创建了一组相关的类、示例和测试，它们可以组合在一起成为`ns-3`模块，以便与现有的`ns-3`模块一起供其他研究人员使用
- 具体步骤：
  - 将`aodvKmeans`复制到`~/tarballs/ns-allinone-3.35/ns-3.35/src`
  - 在`~/tarballs/ns-allinone-3.35/ns-3.35`路径下使用终端创建项目并编译

```
sudo ./utils/create-module.py aodvKmeans
sudo ./ns-3.35 configure --enable-examples --enable-tests
sudo ./ns-3.35 build
sudo ./test.py
```

  - 运行实验代码，观察现象

```
sudo ./scratch/aodvKmeansExample.cc
```

  - 修改节点数及`pps`多次重复实验
  - 计算延迟和数据包传输率：遍历`stats`中的每个流，统计发送的数据包数(`txPackets`)、接收的数据包数(`rxPackets`)、丢失的数据包数(`txPackets - rxPackets`)、总的接收字节数(`rxBytes`)、和延迟总和(`delaySum`)。
  - 计算平均延迟：延迟的计算方式是将所有流的延迟总和除以接收的数据包数。
  - 计算数据包传输率:接收的数据包数除以发送的数据包数

五、实验方法

1. 原始AODV路由算法的实现

1.1 算法概述

- `AODV（Ad Hoc On-Demand Distance Vector）`路由协议是一种按需路由协议，专为无线自组网络设计。其主要特点是在需要时动态建立路由。`AODV`使用`RREQ（Route Request）`和`RREP（Route Reply）`消息进行路由发现，并通过`RERR（Route Error）`消息维护路由表的有效性。

1.2 实现细节

- **路由请求（RREQ）**：
  - 当源节点需要与目标节点通信且没有现成的路由时，它会广播`RREQ`消息。
  - 每个节点接收`RREQ`后，检查是否为目标节点或是否有到目标节点的有效路由。
  - 如果是目标节点或有有效路由，则发送RREP消息返回源节点；否则，继续广播`RREQ`消息。
- **路由回复（RREP）**：
  - 当目标节点或有有效路由的中间节点接收到`RREQ`后，生成`RREP`消息并发送回源节点。
  - `RREP`消息沿着从目标节点到源节点的反向路径返回，并在途中更新每个中间节点的路由表。
- **路由维护**：
  - 使用`HELLO`消息检测链路状态，确保路由表中的条目有效。
  - 当检测到链路断裂时，发送`RERR`消息通知受影响的节点，触发新的路由发现过程。

- **路由删除**：
  - 删除不再使用的无效路由条目，释放网络资源。

1.3 代码示例（此处仅展示核心算法）

1.3.1 路由表条目类（`RoutingTableEntry`）

- `RoutingTableEntry` 类表示路由表中的一个条目，包含路由相关信息以及一些辅助操作方法

```
class RoutingTableEntry
{
public:
  RoutingTableEntry (Ptr<NetDevice> dev = 0,Ipv4Address dst = Ipv4Address (),
bool vSeqNo = false, uint32_t seqNo = 0,
                     Ipv4InterfaceAddress iface = Ipv4InterfaceAddress (),
uint16_t  hops = 0,
                     Ipv4Address nextHop = Ipv4Address (), Time lifetime =
Simulator::Now ());

  ~RoutingTableEntry ();
  bool InsertPrecursor (Ipv4Address id);
  bool LookupPrecursor (Ipv4Address id);
  bool DeletePrecursor (Ipv4Address id);
  void DeleteAllPrecursors ();
  bool IsPrecursorListEmpty () const;
  void GetPrecursors (std::vector<Ipv4Address> & prec) const;
  void Invalidate (Time badLinkLifetime);
  Ipv4Address GetDestination () const
  {
    return m_ipv4Route->GetDestination ();
  }
  Ptr<Ipv4Route> GetRoute () const
  {
    return m_ipv4Route;
  }
  void SetRoute (Ptr<Ipv4Route> r)
  {
    m_ipv4Route = r;
  }
  void SetNextHop (Ipv4Address nextHop)
  {
    m_ipv4Route->SetGateway (nextHop);
  }
  Ipv4Address GetNextHop () const
  {
    return m_ipv4Route->GetGateway ();
  }
  void SetOutputDevice (Ptr<NetDevice> dev)
  {
    m_ipv4Route->SetOutputDevice (dev);
  }
  Ptr<NetDevice> GetOutputDevice () const
  {
    return m_ipv4Route->GetOutputDevice ();
  }
```

```
Ipv4InterfaceAddress GetInterface () const
{
  return m_iface;
}
void SetInterface (Ipv4InterfaceAddress iface)
{
  m_iface = iface;
}

void SetValidSeqNo (bool s)
{
  m_validSeqNo = s;
}
bool GetValidSeqNo () const
{
  return m_validSeqNo;
}
void SetSeqNo (uint32_t sn)
{
  m_seqNo = sn;
}
uint32_t GetSeqNo () const
{
  return m_seqNo;
}
void SetHop (uint16_t hop)
{
  m_hops = hop;
}
uint16_t GetHop () const
{
  return m_hops;
}
void SetLifeTime (Time lt)
{
  m_lifeTime = lt + Simulator::Now ();
}
Time GetLifeTime () const
{
  return m_lifeTime - Simulator::Now ();
}
void SetFlag (RouteFlags flag)
{
  m_flag = flag;
}
RouteFlags GetFlag () const
{
  return m_flag;
}
void SetRreqCnt (uint8_t n)
{
  m_reqCount = n;
}
uint8_t GetRreqCnt () const
{
  return m_reqCount;
```

```
  }
  void IncrementRreqCnt ()
  {
    m_reqCount++;
  }
  void SetUnidirectional (bool u)
  {
    m_blackListState = u;
  }
  bool IsUnidirectional () const
  {
    return m_blackListState;
  }
  void SetBlacklistTimeout (Time t)
  {
    m_blackListTimeout = t;
  }
  Time GetBlacklistTimeout () const
  {
    return m_blackListTimeout;
  }
  Timer m_ackTimer;
  bool operator== (Ipv4Address const  dst) const
  {
    return (m_ipv4Route->GetDestination () == dst);
  }
  void Print (Ptr<OutputStreamWrapper> stream, Time::Unit unit = Time::S) const;

private:
  bool m_validSeqNo;
  uint32_t m_seqNo;
  uint16_t m_hops;
  Time m_lifeTime;
  Ptr<Ipv4Route> m_ipv4Route;
  Ipv4InterfaceAddress m_iface;
  RouteFlags m_flag;
  std::vector<Ipv4Address> m_precursorList;
  Time m_routeRequestTimout;
  uint8_t m_reqCount;
  bool m_blackListState;
  Time m_blackListTimeout;
};
```

- 构造函数

```
RoutingTableEntry::RoutingTableEntry (Ptr<NetDevice> dev, Ipv4Address dst, bool
vSeqNo, uint32_t seqNo,
                                      Ipv4InterfaceAddress iface, uint16_t hops,
Ipv4Address nextHop, Time lifetime)
  : m_ackTimer (Timer::CANCEL_ON_DESTROY),
    m_validSeqNo (vSeqNo),
    m_seqNo (seqNo),
    m_hops (hops),
    m_lifeTime (lifetime + Simulator::Now ()),
```

```
    m_iface (iface),
    m_flag (VALID),
    m_reqCount (0),
    m_blackListState (false),
    m_blackListTimeout (Simulator::Now ())
{
  m_ipv4Route = Create<Ipv4Route> ();
  m_ipv4Route->SetDestination (dst);
  m_ipv4Route->SetGateway (nextHop);
  m_ipv4Route->SetSource (m_iface.GetLocal ());
  m_ipv4Route->SetOutputDevice (dev);
}
```

- 该构造函数初始化了路由表条目的各个成员变量。重要的变量包括：
  - `m_validSeqNo`：是否有效的序列号。
  - `m_seqNo`：序列号。
  - `m_hops`：跳数。
  - `m_lifeTime`：路由条目的生存时间。
  - `m_iface`：接口地址。
  - `m_flag`：路由条目状态。
  - `m_reqCount`：请求计数。
  - `m_blackListState`：黑名单状态。
  - `m_blackListTimeout`：黑名单超时时间。
  - `m_ipv4Route`：IPv4路由对象。
- 核心方法
  - `InsertPrecursor`：插入前驱节点。
  - `LookupPrecursor`：查找前驱节点。
  - `DeletePrecursor`：删除前驱节点。
  - `DeleteAllPrecursors`：删除所有前驱节点。
  - `IsPrecursorListEmpty`：检查前驱列表是否为空。
  - `GetPrecursors`：获取前驱列表。
  - `Invalidate`：使路由条目失效。
  - `Print`：打印路由条目信息。

1.3.2 路由表类（`RoutingTable`）

- `RoutingTable` 类管理所有路由条目，提供路由查找、添加、删除和更新等功能。

```
class RoutingTable
{
public:
  RoutingTable (Time t);
  Time GetBadLinkLifetime () const
  {
    return m_badLinkLifetime;
  }
```

```
  void SetBadLinkLifetime (Time t)
  {
    m_badLinkLifetime = t;
  }
  bool AddRoute (RoutingTableEntry & r);
  bool DeleteRoute (Ipv4Address dst);
  bool LookupRoute (Ipv4Address dst, RoutingTableEntry & rt);
  bool LookupValidRoute (Ipv4Address dst, RoutingTableEntry & rt);
  bool Update (RoutingTableEntry & rt);
  bool SetEntryState (Ipv4Address dst, RouteFlags state);
  void GetListOfDestinationWithNextHop (Ipv4Address nextHop,
std::map<Ipv4Address, uint32_t> & unreachable);
  void InvalidateRoutesWithDst (std::map<Ipv4Address, uint32_t> const &
unreachable);
  void DeleteAllRoutesFromInterface (Ipv4InterfaceAddress iface);
  void Clear ()
  {
    m_ipv4AddressEntry.clear ();
  }
  void Purge ();
  bool MarkLinkAsUnidirectional (Ipv4Address neighbor, Time blacklistTimeout);
  void Print (Ptr<OutputStreamWrapper> stream, Time::Unit unit = Time::S) const;

private:
  std::map<Ipv4Address, RoutingTableEntry> m_ipv4AddressEntry;
  Time m_badLinkLifetime;
  void Purge (std::map<Ipv4Address, RoutingTableEntry> &table) const;
};
```

- 构造函数

```
RoutingTable::RoutingTable (Time t)
  : m_badLinkLifetime (t)
{
}
```

构造函数初始化了路由表，并设置了坏链路生存时间。

- 方法
  - `LookupRoute`：查找路由。
  - `LookupValidRoute`：查找有效路由。
  - `DeleteRoute`：删除路由。
  - `AddRoute`：添加路由。
  - `Update`：更新路由。
  - `SetEntryState`：设置路由条目状态。
  - `GetListOfDestinationWithNextHop`：获取具有相同下一跳的目的地址列表。
  - `InvalidateRoutesWithDst`：使具有特定目的地址的路由失效。
  - `DeleteAllRoutesFromInterface`：删除所有从特定接口来的路由。
  - `Purge`：清除无效路由。

- `MarkLinkAsUnidirectional`：标记链路为单向。
- `Print`：打印路由表。

- 路由查找方法

```
bool
RoutingTable::LookupRoute (Ipv4Address id, RoutingTableEntry & rt)
{
  NS_LOG_FUNCTION (this << id);
  Purge ();
  if (m_ipv4AddressEntry.empty ())
    {
      NS_LOG_LOGIC ("Route to " << id << " not found; m_ipv4AddressEntry is
empty");
      return false;
    }
  std::map<Ipv4Address, RoutingTableEntry>::const_iterator i =
    m_ipv4AddressEntry.find (id);
  if (i == m_ipv4AddressEntry.end ())
    {
      NS_LOG_LOGIC ("Route to " << id << " not found");
      return false;
    }
  rt = i->second;
  NS_LOG_LOGIC ("Route to " << id << " found");
  return true;
}
```

`LookupRoute` 方法在路由表中查找目标地址为 `id` 的路由条目。如果找到，则返回 `true` 并将路由条目赋值给 `rt`；否则，返回 `false`。

- 路由添加方法

```
bool
RoutingTable::AddRoute (RoutingTableEntry & rt)
{
  NS_LOG_FUNCTION (this);
  Purge ();
  if (rt.GetFlag () != IN_SEARCH)
    {
      rt.SetRreqCnt (0);
    }
  std::pair<std::map<Ipv4Address, RoutingTableEntry>::iterator, bool> result =
    m_ipv4AddressEntry.insert (std::make_pair (rt.GetDestination (), rt));
  return result.second;
}
```

`AddRoute` 方法将一个新的路由条目添加到路由表中。如果条目不是处于搜索状态，则将请求计数设置为0。返回值表示是否添加成功。

- 路由更新方法

```
bool
RoutingTable::Update (RoutingTableEntry & rt)
```

```
{
  NS_LOG_FUNCTION (this);
  std::map<Ipv4Address, RoutingTableEntry>::iterator i =
    m_ipv4AddressEntry.find (rt.GetDestination ());
  if (i == m_ipv4AddressEntry.end ())
    {
      NS_LOG_LOGIC ("Route update to " << rt.GetDestination () << " fails; not
found");
      return false;
    }
  i->second = rt;
  if (i->second.GetFlag () != IN_SEARCH)
    {
      NS_LOG_LOGIC ("Route update to " << rt.GetDestination () << " set RreqCnt
to 0");
      i->second.SetRreqCnt (0);
    }
  return true;
}
```

`Update` 方法更新路由条目。如果找到目标地址对应的路由条目，则更新其信息并返回 `true`；否则，返回 `false`。

- 路由清理方法

```
void
RoutingTable::Purge ()
{
  NS_LOG_FUNCTION (this);
  if (m_ipv4AddressEntry.empty ())
    {
      return;
    }
  for (std::map<Ipv4Address, RoutingTableEntry>::iterator i =
         m_ipv4AddressEntry.begin (); i != m_ipv4AddressEntry.end (); )
    {
      if (i->second.GetLifeTime () < Seconds (0))
        {
          if (i->second.GetFlag () == INVALID)
            {
              std::map<Ipv4Address, RoutingTableEntry>::iterator tmp = i;
              ++i;
              m_ipv4AddressEntry.erase (tmp);
            }
          else if (i->second.GetFlag () == VALID)
            {
              NS_LOG_LOGIC ("Invalidate route with destination address " << i-
>first);
              i->second.Invalidate (m_badLinkLifetime);
              ++i;
            }
          else
            {
              ++i;
            }
```

```
        }
      else
        {
          ++i;
        }
    }
}
```

`Purge` 方法清理路由表中无效的路由条目。如果条目的生存时间小于0并且状态为无效，则将其从路由表中删除；如果状态为有效，则使其失效并更新生存时间。

- 路由标记方法

```
bool
RoutingTable::MarkLinkAsUnidirectional (Ipv4Address neighbor, Time
blacklistTimeout)
{
  NS_LOG_FUNCTION (this << neighbor << blacklistTimeout.As (Time::S));
  std::map<Ipv4Address, RoutingTableEntry>::iterator i =
    m_ipv4AddressEntry.find (neighbor);
  if (i == m_ipv4AddressEntry.end ())
    {
      NS_LOG_LOGIC ("Mark link unidirectional to  " << neighbor << " fails; not
found");
      return false;
    }
  i->second.SetUnidirectional (true);
  i->second.SetBlacklistTimeout (blacklistTimeout);
  i->second.SetRreqCnt (0);
  NS_LOG_LOGIC ("Set link to " << neighbor << " to unidirectional");
  return true;
}
```

`MarkLinkAsUnidirectional` 方法将指定的邻居标记为单向链路，并设置黑名单超时时间。

1.3.3 总结

- 以上代码实现了AODV路由协议中的核心功能，包括路由表条目管理和路由表操作。路由表条目类（`RoutingTableEntry`）负责存储每个路由的详细信息，并提供对前驱节点的操作方法。路由表类（`RoutingTable`）提供路由查找、添加、更新和删除等基本功能，同时支持对路由条目的清理和标记操作。通过这些核心算法，AODV协议能够动态地管理和维护无线自组网络中的路由信息。

2. 基于K-Means聚类的AODV优化

2.1 K-Means算法概述

`K-Means`聚类算法是一种无监督学习算法，用于将数据集划分为K个簇。其目标是最小化簇内数据点之间的总平方误差（`SSE`）。算法步骤包括初始化K个聚类中心、分配数据点到最近的聚类中心、更新聚类中心以及重复以上步骤直到收敛。

2.2 优化方法设计

为了优化`AODV`路由协议，本项目设计了一种基于`K-Means`聚类算法的优化方法，具体步骤如下：

1. **节点聚类**：
   - 使用`K-Means`算法将网络中的节点聚类，确定K个聚类中心节点。
   - 每个中心节点作为该聚类的主要`RREQ`转发器，减少泛洪范围。
2. **路由请求（`RREQ`）优化**：
   - 源节点发送`RREQ`消息时，首先发送给其所属聚类的中心节点。
   - 中心节点负责将`RREQ`消息转发给目标节点或其他中心节点，减少不必要的控制包传输。
3. **路由回复（`RREP`）优化**：
   - `RREP`消息通过中心节点返回源节点，同样减少控制包的泛洪。
4. **路由维护和删除**：
   - 采用与原始`AODV`类似的方法进行路由维护和删除，但减少了控制消息的泛洪范围。

2.3 实现细节

1. **初始化节点聚类**：
   - 在网络初始化阶段，使用K-Means算法对所有节点进行聚类，确定每个节点的聚类中心。
2. **优化RREQ转发**：
   - 源节点发送`RREQ`消息时，首先发送给其聚类中心。
   - 聚类中心接收到`RREQ`消息后，检查是否为目标节点或是否有到目标节点的路由。
   - 如果是，则发送`RREP`消息；否则，中心节点将RREQ消息转发给其他相关中心节点。
3. **优化RREP传输**：
   - `RREP`消息沿着中心节点返回源节点，减少不必要的中间节点参与。
4. **路由维护**：
   - 通过定期发送`HELLO`消息维护路由表，确保路由有效性。
   - 检测到链路断裂时，发送`RERR`消息，并由中心节点进行路由重新发现。

2.4 代码示例（此处仅展示核心算法）

1. 函数签名

```
std::vector<Ipv4Address>
RoutingTable::Kmeans (Ipv4Address dst, uint32_t positionX, uint32_t positionY)
```

- 该函数返回一个包含IPv4地址的向量，这些地址代表了最佳的RREQ转发节点集群。输入参数包括目标节点的IPv4地址`dst`、当前节点的X坐标`positionX`和Y坐标`positionY`。

2. 数据清理

```
Purge();
```

- 调用`Purge()`函数清理路由表中无效的条目。

3. 特征提取与初始化

```
int n = m_ipv4AddressEntry.size();
double features[n][3];
int k = 2;
double cluster_center[k][3];
int cluster_assignments[n];

int i = 0;
for (std::map<Ipv4Address, RoutingTableEntry>::iterator it =
m_ipv4AddressEntry.begin (); it != m_ipv4AddressEntry.end (); ++it)
{
    if(it->first.IsBroadcast() || it->first.IsLocalhost() || it-
>first.IsMulticast() ||
    it->first.IsSubnetDirectedBroadcast(Ipv4Mask((char *)"255.255.255.0")) || it-
>second.GetFlag() == INVALID || it->second.GetHop() > 2)
    {
        continue;
    }
    features[i][0] = 1.0 * (positionX - it->second.GetPositionX()) * (positionX -
it->second.GetPositionX()) +
                     1.0 * (positionY - it->second.GetPositionY()) * (positionY -
it->second.GetPositionY());
    features[i][1] = 1.0 * it->second.GetTxErrorCount();
    features[i][2] = 1.0 * it->second.GetFreeSpace();
    i++;
}
n = i;
```

- 提取每个节点的特征数据，包括到当前节点的距离、传输错误计数和空闲空间。过滤掉广播、局域、组播、子网广播地址以及无效条目和跳数大于2的条目。

4. 特征归一化

```
double mini[3], maxi[3];

for(int j=0;j<3;j++)
{
    mini[j] = features[0][j];
    maxi[j] = features[0][j];
    for(int i=1;i<n;i++)
    {
        mini[j] = std::min(mini[j], features[i][j]);
        maxi[j]= std::max(maxi[j], features[i][j]);
    }
    for(int i=0;i<n;i++)
    {
        features[i][j] = (features[i][j] - mini[j]);
        if(mini[j] != maxi[j]) features[i][j] /= (maxi[j] - mini[j]);
    }
}
```

- 对特征进行归一化处理，使特征值在0到1之间，以便于后续的K-Means聚类。

5. 初始化K-Means聚类中心

```
double ideal[3];
ideal[0] = 0.0; // minimum possible distance
ideal[1] = 0.0; // minimum possible error
ideal[2] = maxi[2]; // buffer empty

srand(time(0));
for(int i=0;i<k;i++)
{
    int p = rand() % n;
    for(int j=0;j<3;j++)
    {
        cluster_center[i][j] = features[p][j];
    }
}
```

- 随机选择初始聚类中心，并定义一个理想的特征向量（最小距离、最小错误和最大空闲空间）。

6. 迭代K-Means聚类

```
int num_iterations = 3;
for(int iteration=0;iteration<=num_iterations;iteration++)
{
    for(int i=0;i<n;i++)
    {
        double mini_dist = -1.0;
        for(int j=0;j<k;j++)
        {
            double dist = 0.0;
            for(int d=0; d<3; d++)
            {
                dist += (features[i][d] - cluster_center[j][d]) * (features[i][d]
- cluster_center[j][d]);
            }
            if(mini_dist == -1.0 || dist < mini_dist)
            {
                mini_dist = dist;
                cluster_assignments[i] = j;
            }
        }
    }
    if(iteration == num_iterations)
    {
        break;
    }
    for(int j=0;j<k;j++)
    {
        int cnt = 0;
        for(int d=0;d<3;d++)
        {
            cluster_center[j][d] = 0.0;
        }
        for(int i=0;i<n;i++)
        {
```

```
            if(cluster_assignments[i] == j)
            {
                for(int d=0;d<3;d++)
                {
                    cluster_center[j][d] += features[i][d];
                }
                cnt++;
            }
        }
        if(cnt)
        {
            for(int d=0;d<3;d++)
            {
                cluster_center[j][d] /= cnt;
            }
        }
        else
        {
            int p = rand() % n;
            for(int d=0;d<3;d++)
            {
                cluster_center[j][d] = features[p][d];
            }
        }
    }
}
```

- 执行K-Means聚类算法，通过指定的迭代次数（默认3次），不断更新聚类中心和节点的聚类分配。

7. 反归一化特征值和聚类中心

```
for(int j=0;j<k;j++)
{
    for(int d=0;d<3;d++)
    {
        cluster_center[j][d] *= (maxi[d] - mini[d]);
        cluster_center[j][d] += mini[d];
    }
}
for(int i=0;i<n;i++)
{
    for(int d=0;d<3;d++)
    {
        features[i][d] *= (maxi[d] - mini[d]);
        features[i][d] += mini[d];
    }
}
```

- 将特征值和聚类中心反归一化，恢复到原始尺度。

8. 选择最佳聚类

```
int optimal_cluster = -1;
double mini_dist = -1.0;
for(int j=0;j<k;j++)
{
    double dist = 0;
    for(int d=0; d<3; d++)
    {
        dist += (cluster_center[j][d] - ideal[d]) * (cluster_center[j][d] -
ideal[d]);
    }
    if(mini_dist == -1.0 || dist < mini_dist)
    {
        mini_dist = dist;
        optimal_cluster = j;
    }
}
```

- 计算每个聚类中心到理想特征向量的距离，选择最接近理想特征向量的聚类作为最佳聚类。

9. 返回最佳聚类中的节点

```
std::vector<Ipv4Address>selectedCluster;
i = 0;
for (std::map<Ipv4Address, RoutingTableEntry>::iterator it =
m_ipv4AddressEntry.begin (); it != m_ipv4AddressEntry.end (); ++it)
{
    if(it->first.IsBroadcast() || it->first.IsLocalhost() || it-
>first.IsMulticast() ||
    it->first.IsSubnetDirectedBroadcast(Ipv4Mask((char *)"255.255.255.0")) || it-
>second.GetFlag() == INVALID || it->second.GetHop() > 2)
    {
        continue;
    }
    if(cluster_assignments[i] == optimal_cluster)
    {
        selectedCluster.push_back(it->first);
    }
    i++;
}
return selectedCluster;
```

- 遍历路由表，将属于最佳聚类的节点的IPv4地址加入结果向量，并返回该向量。

10. 总结

- 该代码实现了一个基于K-Means聚类算法的节点选择方法，用于优化AODV路由协议的RREQ转发过程。通过提取节点的距离、传输错误计数和空闲空间等特征，对节点进行聚类，并选择最佳的聚类用于转发RREQ数据包，从而减少网络中的控制包数量，提升网络性能。

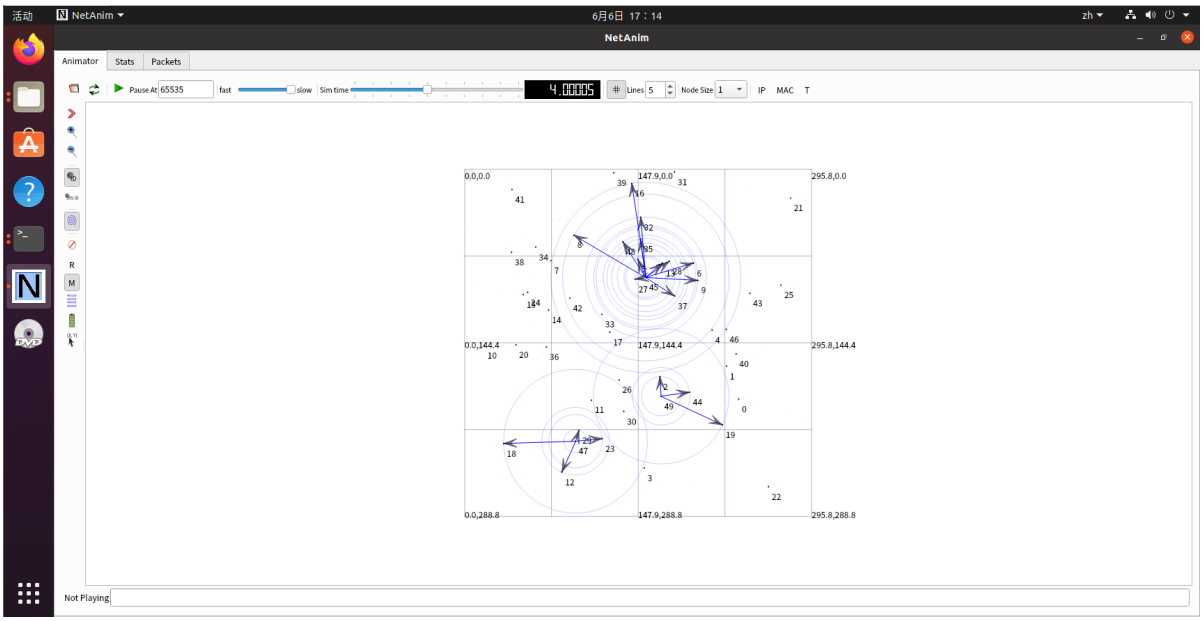
五、实验结果

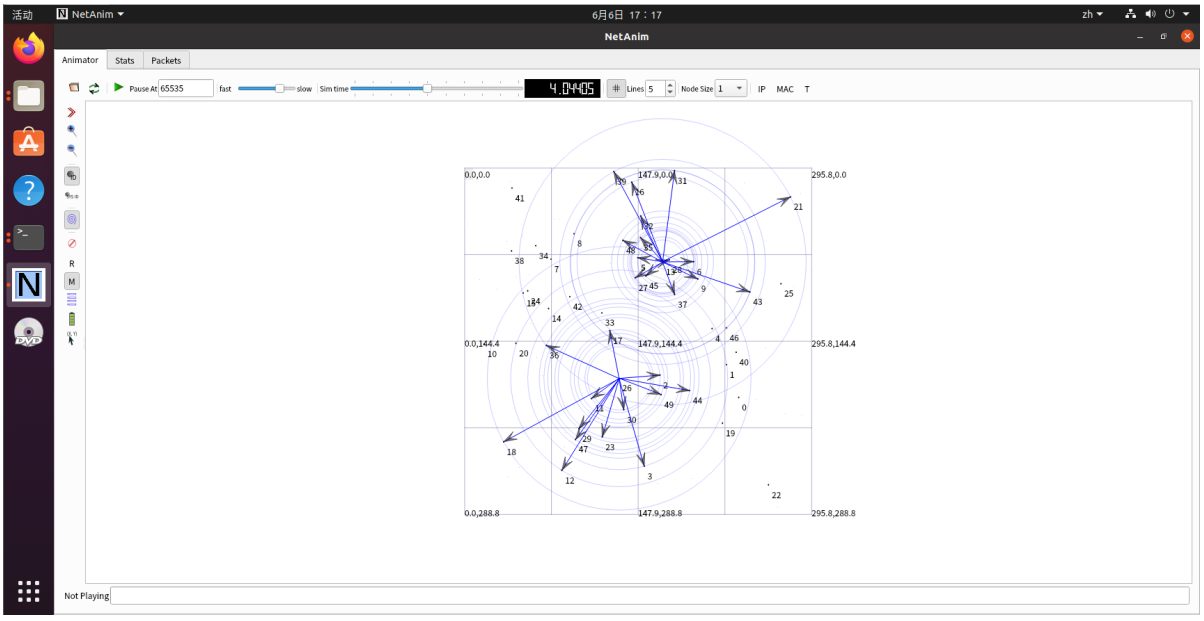
- 观察结点间包的发送
    - 从`NetAnim`截图中可以观察到以下几点：

1. **节点分布**：
    - 节点在一个二维平面上随机分布，覆盖了一个较大的区域。
    - 节点之间的距离不均匀，有些节点之间距离较近，而有些节点之间距离较远。
2. **节点连线**：
    - 图中显示了节点之间的通信路径，连线的箭头表示数据传输的方向。
    - 部分节点之间有多条连线，表明这些节点之间有较多的数据交换。
    - 连线的颜色和粗细可能表示不同的通信状态或数据传输量。
3. **聚类特征**：
    - 从图中可以看到几个明显的聚类现象，每个簇内的节点之间连线较密集，簇与簇之间的连线相对较少。
    - 这种聚类可能是基于节点的地理位置、信号强度或其他特征进行的。
4. **箭头和连线**：
    - 图中的箭头和连线表示节点之间的通信方向和路径。
    - 蓝色箭头表示数据包的发送方向，从一个节点指向另一个节点。
    - 连线的存在和方向说明了节点之间的通信链路。

- 该图展示了基于K-Means聚类算法的AODV路由协议的仿真结果。图中的节点通过K-Means算法被分成了几个聚类簇，每个簇内的节点之间的通信较为密集，簇间通信相对较少。这种结果说明K-Means聚类算法有效地将相邻或通信频繁的节点分配到同一个簇中，从而减少了不必要的RREQ（路由请求）广播，优化了AODV路由协议的性能。具体解释如下：
    1. **聚类效果**：
        - 图中的节点被分成了几个聚类，每个簇内的节点之间有较多的通信路径，说明聚类算法有效地将通信频繁的节点聚集在一起。
        - 聚类的边界较为清晰，每个簇内部的连线较多，簇之间的连线相对较少。
    2. **通信路径**：
        - 每个节点通过箭头指示的路径向其他节点发送数据包。
        - 聚类内的节点之间直接通信，减少了跨簇通信的需求，从而减少了网络的拥塞和延迟。
    3. **优化效果**：
        - 通过K-Means聚类，AODV路由协议减少了RREQ广播的范围，提高了路由发现的效率。
        - 聚类内的通信频率高，簇间通信频率低，减少了不必要的数据传输，提高了网络性能。

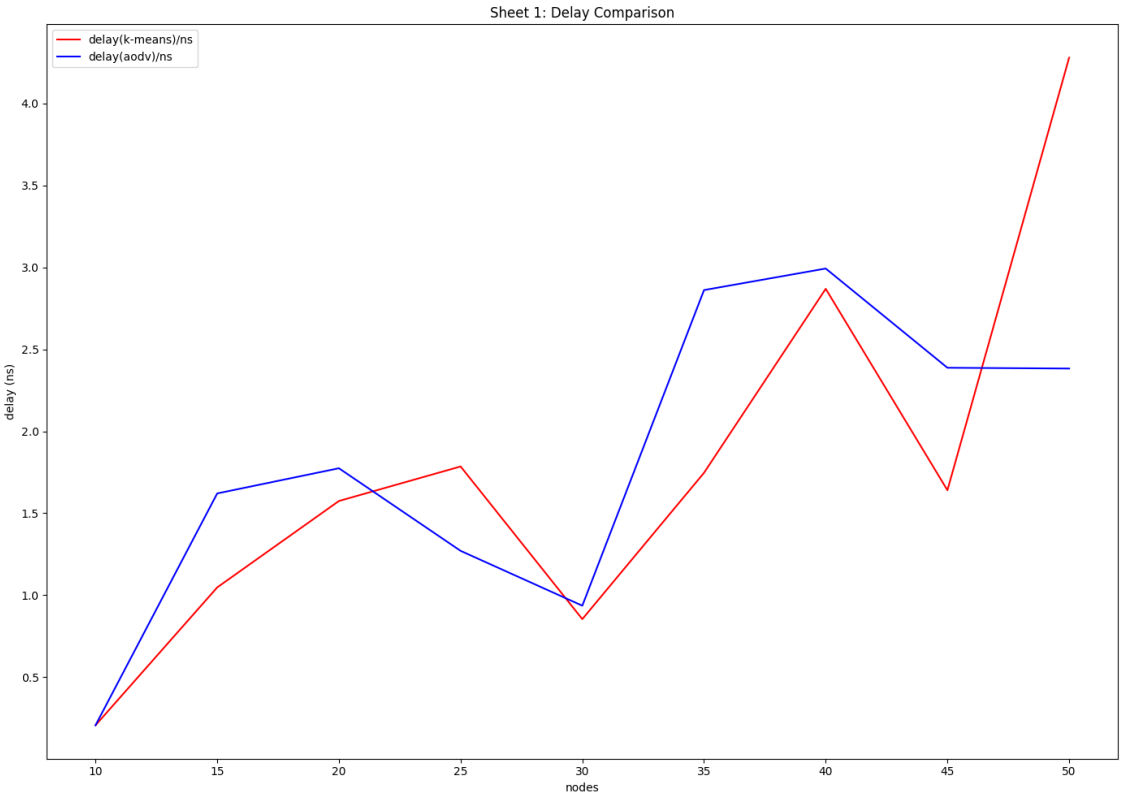
- 总的来说，这张图展示了K-Means聚类算法在优化AODV路由协议中的应用效果，有效地减少了网络拥塞和延迟，提高了数据传输的效率。





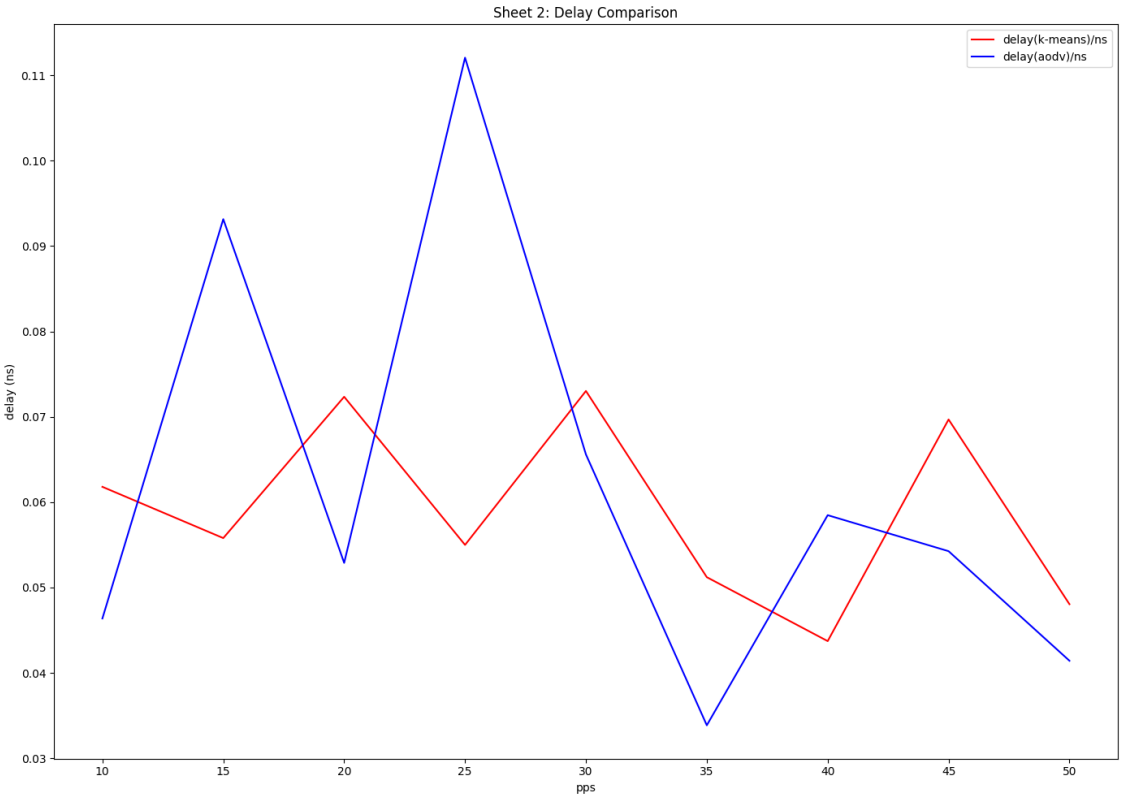
- `aodvKmeans`聚类算法与`aodv`原始算法对比分析

  1. **节点数量与延迟对比**`(Delay vs. Nodes)`



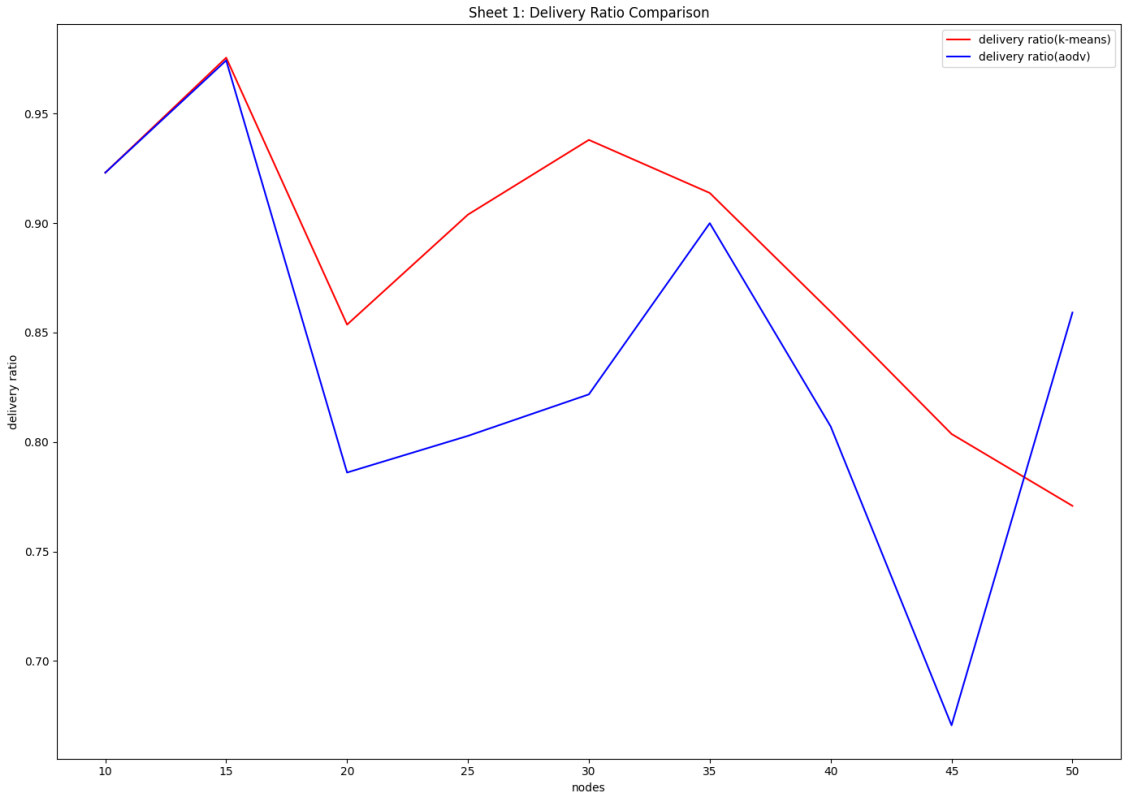
- **X轴**：节点数量（nodes）。
- **Y轴**：延迟（delay），单位为纳秒（ns）。
- **红线**：K-Means优化的AODV路由算法的延迟。
- **蓝线**：原始AODV路由算法的延迟。
- 分析：
  - 随着节点数量的增加，两个算法的延迟都呈现波动。
  - 对于大多数节点数量，K-Means优化的AODV算法的延迟略低于原始AODV算法，说明K-Means优化在大多数情况下能够降低网络延迟。
  - 在某些节点数量（例如25和50）下，原始AODV算法的延迟显著高于K-Means优化的AODV算法。

2. **每秒数据包数与延迟对比 (Delay vs. PPS)**



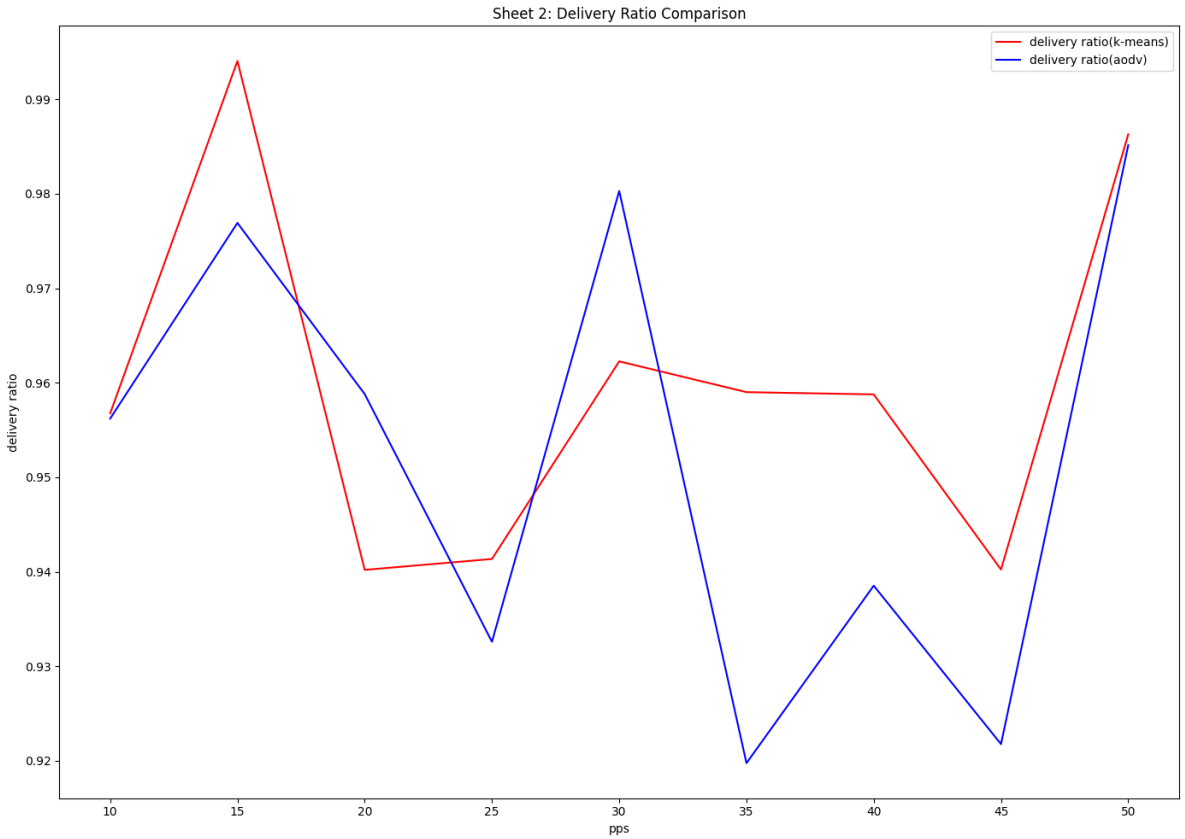
- **X轴**：每秒数据包数（PPS）。
- **Y轴**：延迟（delay），单位为纳秒（ns）。
- **红线**：K-Means优化的AODV路由算法的延迟。
- **蓝线**：原始AODV路由算法的延迟。
- 分析：
  - 随着每秒数据包数的增加，两个算法的延迟都呈现波动。
  - 总体来看，K-Means优化的AODV算法在大多数情况下延迟较低，尤其是在数据包数较多时（例如30和50）。
  - K-Means优化的AODV算法在某些数据包数下（例如20和30）的延迟略高于原始AODV算法，可能是由于聚类算法带来的计算开销。

3. **节点数量与数据包传输率对比 (Delivery Ratio vs. Nodes)**



  - **X轴**：节点数量（nodes）。
- **Y轴**：数据包传输率（delivery ratio），单位为比例。
  - **红线**：K-Means优化的AODV路由算法的数据包传输率。
- **蓝线**：原始AODV路由算法的数据包传输率。
  - 分析
    - 随着节点数量的增加，两个算法的数据包传输率都呈现波动。
- 对于大多数节点数量，K-Means优化的AODV算法的数据包传输率高于原始AODV算法，说明K-Means优化在大多数情况下能够提高数据包传输率。

4. **每秒数据包数与数据包传输率对比 (Delivery Ratio vs. PPS)**



- **X轴**：每秒数据包数（PPS）。
- **Y轴**：数据包传输率（delivery ratio），单位为比例。
- **红线**：K-Means优化的AODV路由算法的数据包传输率。
- **蓝线**：原始AODV路由算法的数据包传输率。
- 分析：
  - 随着每秒数据包数的增加，两个算法的数据包传输率都呈现波动。
  - 总体来看，K-Means优化的AODV算法在大多数情况下的数据包传输率较高，尤其是在数据包数较多时（例如30和50）。
  - K-Means优化的AODV算法在某些数据包数下（例如20和30）略低于原始AODV算法，但总体差距不大。

六、 实验反思

1. 聚类中的距离是物理距离吗？

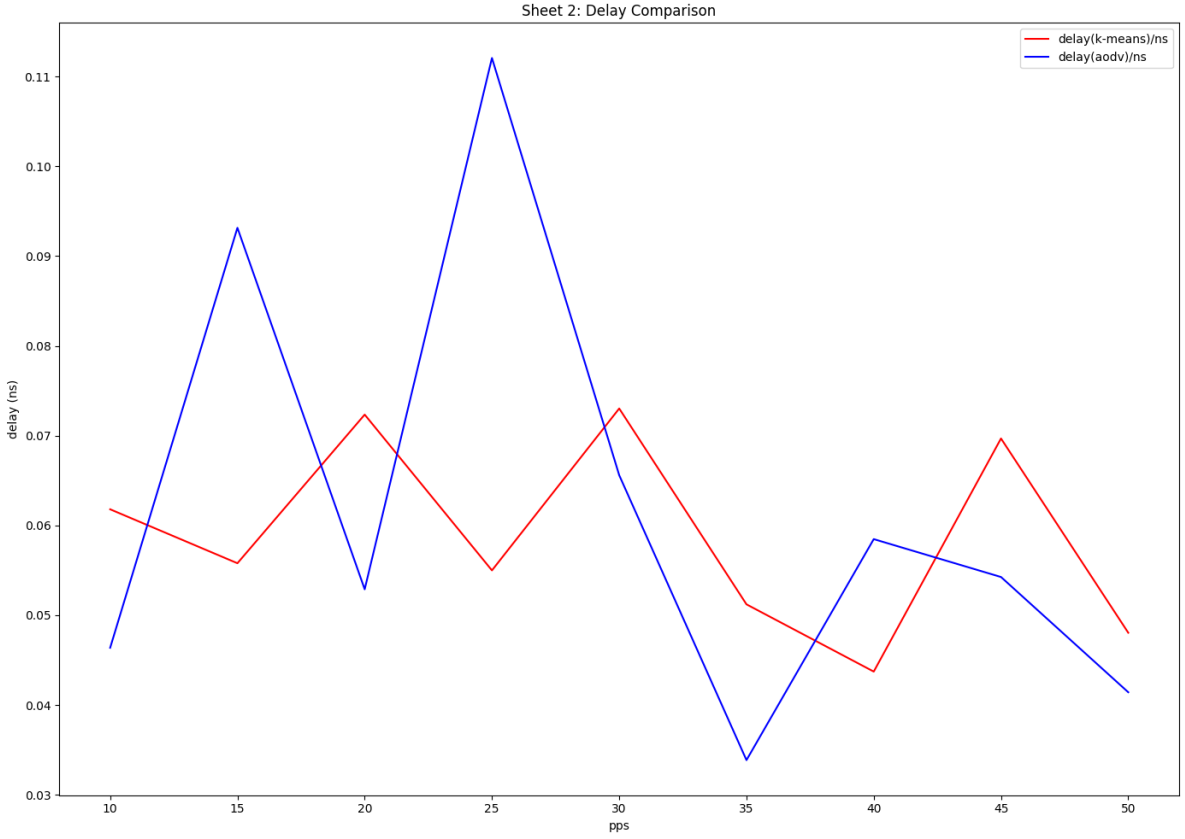
- 是的，在网络仿真中，特别是像NS3这样的网络仿真器中，K-Means算法通常使用的是物理距离，即节点之间的地理空间距离。这种距离度量方式最符合实际应用场景中的网络拓扑结构。
- 物理距离 (Physical Distance)
  - 物理距离在网络仿真中通常指节点在二维或三维空间中的直线距离。在K-Means聚类中，物理距离可以用欧几里得距离来表示，具体如下：
  - 欧几里得距离 (Euclidean Distance)
    - 在二维空间中，两个节点 (x_1, y_1) 和 (x_2, y_2)之间的欧几里得距离计算公式为：

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2}$$

- 在三维空间中，两个节点 (x_1, y_1, z_1) 和 (x_2, y_2, z_2)之间的欧几里得距离计算公式为：

$$d = \sqrt{(x_2 - x_1)^2 + (y_2 - y_1)^2 + (z_2 - z_1)^2}$$

2. Delay v.s. pps图中延迟降低的原因？



- 从上图中可以看出，蓝色线（原始AODV算法）的延迟随`pps`（每秒分组数）的增加而降低，这种现象可能与以下几个原因有关：
  - **网络拥塞**：在低`pps`时，网络可能更容易发生拥塞，导致数据包排队等待时间增加，从而增加延迟。当`pps`增加时，网络中的数据流变得更加均衡，减少了拥塞现象，从而降低延迟。
  - **路由缓存**：AODV算法具有路由缓存机制。在低`pps`情况下，路由请求频繁，导致路由发现过程消耗时间较多，增加延迟。而在高`pps`情况下，路由缓存可能被更频繁地使用，减少了路由发现的时间，从而降低了延迟。
  - **信道利用率**：在低`pps`时，信道利用率可能较低，导致通信节点之间的竞争较少，频繁的路由更新和通信中断增加了延迟。随着`pps`增加，信道利用率提高，通信节点之间的竞争更加平衡，减少了延迟。
  - **数据包大小和频率**：数据包的大小和发送频率可能影响网络的延迟。在高`pps`时，较小的数据包更容易通过网络传输，减少了单个数据包的传输时间，降低了整体延迟。

3. 计算开销？K-means在cluster head计算吗？计算延迟？

- **聚类的开销**主要包括以下几个方面：
  - **计算开销**：每次迭代中，计算每个节点到各个质心的距离，并更新质心位置的计算量。
  - **通信开销**：在分布式环境中，需要各节点与中心节点（或聚类头节点）交换信息，通信开销取决于网络拓扑和数据交换频率。
  - **存储开销**：需要存储节点的位置、当前聚类分配和质心位置等信息。

- **K-Means在Cluster Head计算**
  - 在典型的K-Means算法中，聚类中心（Cluster Head）的计算可以集中进行，也可以分布式进行。以下是两种情况的说明：
  - **集中计算**：
    - 所有节点将自己的位置数据发送到一个中心节点或服务器。
    - 中心节点或服务器计算所有节点的聚类，并返回聚类结果给各节点。
    - 优点：计算简单，易于实现。
    - 缺点：通信开销大，中心节点成为单点故障。
  - **分布式计算**：
    - 各节点或局部区域内的节点协同计算，选举出本地的聚类中心。
    - 每个聚类中心与其他中心协同，逐步收敛到全局聚类。
    - 优点：分散计算和通信开销，增强系统的鲁棒性。
    - 缺点：实现复杂，算法收敛速度可能较慢。
- **计算延迟的位置**
  - **延迟计算**可以在以下几个地方进行：
    - **在各节点本地计算**：每个节点自己计算与聚类中心的距离，并选择最近的聚类中心。这种方式的优点是可以减少中心节点的计算负担，但缺点是需要节点具有较强的计算能力。
    - **在中心节点计算**：所有节点将自己的位置数据发送到中心节点，中心节点计算所有节点的聚类分配。这种方式简单，但容易造成中心节点的计算和通信负载过高。

在我们的项目中我们均采用集中计算，原因如下：

- 集中计算相对于分布式计算来说，算法实现更为简单。所有的计算任务由一个中心节点完成，减少了节点间的通信复杂度和协调问题，避免了在分布式计算中可能出现的同步和一致性问题。在NS3网络仿真环境中，集中计算可以更容易地管理和控制仿真过程。中心节点可以收集所有节点的位置数据，进行聚类计算，并将结果返回给各个节点。这种方式可以简化仿真过程的管理，特别是在需要多次迭代和调试的情况下。集中计算减少了节点间的通信开销。在分布式计算中，节点需要频繁交换位置信息和聚类状态，这会增加网络负载和通信延迟。集中计算则将所有通信集中到中心节点，减少了整体的通信量。
- 集中计算能够保证聚类计算的一致性和准确性。由于所有计算都在一个节点上完成，避免了分布式计算中可能出现的节点间数据不一致的问题。这对于需要高精度和一致性的仿真结果尤为重要。

七、总结

- 本研究提出并实现了一种基于K-Means聚类算法的AODV路由协议优化方法，并在NS3仿真平台上对其进行了验证。传统的AODV路由协议在进行路由发现时采用泛洪方法，虽然简单高效，但在高密度网络中会产生大量的控制数据包，导致网络资源浪费和网络拥塞。为了克服这些不足，本研究引入了K-Means聚类算法，通过聚类方法优化路由发现过程，减少了控制数据包的传输量，提升了网络性能。
- **主要研究贡献如下：**
  - **引入K-Means聚类算法**：利用K-Means聚类算法对网络节点进行聚类，选择各个集群的中心节点作为路由请求（RREQ）的主要转发节点。这种方法减少了不必要的RREQ数据包传输，降低了网络拥塞和控制开销。
  - **优化AODV路由协议**：在原始AODV协议的基础上，设计并实现了基于K-Means聚类的优化算法。优化后的算法通过减少泛洪范围，提高了路由发现的效率，降低了端到端延迟。

    - **实验验证与性能评估**：通过在NS3仿真平台上的实验，比较了原始AODV和优化AODV的性能。实验结果显示，优化后的AODV协议在高密度网络环境下表现出更优的性能，具体体现在以下几个方面：
        - **延迟**：K-Means优化的AODV算法在大多数情况下能够显著降低网络延迟，尤其是在节点数量和每秒数据包数较多时，优化效果更加明显。
        - **数据包传输率**：优化算法提高了数据包的传输率，减少了数据包丢失，增强了网络的可靠性和稳定性。
        - **控制开销**：通过聚类算法的应用，减少了控制数据包的传输量，从而降低了网络的总体控制开销，提高了网络资源的利用率。
    - **适应性与鲁棒性**：优化后的AODV算法在不同的网络负载和节点分布条件下都表现出较好的适应性和鲁棒性，能够有效应对网络拓扑的动态变化。
- **未来研究方向**：
    - 本研究的优化方法在提升AODV路由协议性能方面取得了显著成果，但仍有一些方面值得进一步研究和探索：
        - **算法复杂度与资源消耗**：尽管优化算法提高了网络性能，但其计算复杂度和资源消耗也有所增加。未来的研究可以进一步优化聚类算法，降低其复杂度，使其在资源受限的环境中也能高效运行。
        - **多种网络环境下的验证**：本研究主要在NS3仿真平台上进行验证，未来可以在更多实际网络环境中进行测试，验证算法的实用性和稳定性。
        - **结合其他机器学习算法**：除了K-Means聚类，还可以探索结合其他机器学习算法，如深度学习、强化学习等，进一步优化路由协议，提高网络性能。
- 综上所述，本研究提出的基于K-Means聚类的AODV路由协议优化方法，通过减少控制数据包传输量和降低网络延迟，有效提升了无线自组织网络的整体性能。未来的研究可以在此基础上，进一步优化算法，拓展其应用范围，推动无线自组织网络技术的发展。