void PropagateThread::split\_twoAjacent\_graph\_next( DualwayPropagation& dp1 ,IndexType srFrame, IndexType tgFrame)

{

QMetaObject::invokeMethod( Global\_Window->getCanvas() ,"updateGL",Qt::QueuedConnection);

//向前分裂

//get the new graph of tgGrame--需要深拷贝

Logger<<" .......\n";

Logger<<" Start next split.\n";

IndexType tgGraSize = dp\_.hier\_componets\_[tgFrame].hier\_graph.size();

LabelsGraph\* oriGra = dp\_.hier\_componets\_[tgFrame].hier\_graph[tgGraSize - 1];

LabelsGraph\* new\_graph = new LabelsGraph(\*oriGra);

//new\_graph = oriGra;

//vector<HLabel\* > new\_label\_bucket = hier\_componets\_[tgFrame].hier\_label\_bucket[tgGraSize - 1];

*vector*<HLabel\* > new\_label\_bucket;

copyLabelBucket(new\_label\_bucket,dp\_.hier\_componets\_[tgFrame].hier\_label\_bucket[tgGraSize - 1] );

//

IndexType gLevel = 0;

IndexType srGraSize = dp\_.hier\_componets\_[srFrame].hier\_graph.size();

gLevel = srGraSize - 1;//获取最新的层

LabelsGraph\* srGraLat = dp\_.hier\_componets\_[srFrame].hier\_graph[gLevel];

IndexType labParentsize = tgGraSize + 1; //生成的层数

Logger<<srFrame<<"帧的第"<<gLevel<<"层边界分割"<<tgFrame<<"的"<<tgGraSize - 1 <<"层"<<endl;

pair<EdgeIterator,EdgeIterator> ei = boost::edges(\*srGraLat);

emit writeSignal(QString("target frame")+QString("%1").arg(tgFrame)+QString("begin"),1); //释放写信号,写到主窗口的输出框中

for (EdgeIterator eit = ei.first; eit != ei.second; ++eit)

{

EdgeDescriptor ed = \*eit;

GraphEdgeProperty& ep = (\*srGraLat)[ed];

map<IndexType,HVertex\*> edgePoints;

auto ePsIt = ep.edgePoints.begin();

edgePoints.insert(ePsIt->second.begin(),ePsIt->second.end() );

if (edgePoints.size() < 3)

{

Logger<<"边上的顶点数太少,无法分裂.\n";

continue;

}

map<IndexType,HVertex\*> edgeCorrNextVtx;

IndexType newGraphEdgeSize = new\_graph->m\_edges.size();

IndexType nodeId = checkNextLabelBucket(edgePoints,edgeCorrNextVtx);//获得边界点在下一帧对应的块和对应点

//IndexType nodeId = edgeCorrNextVtx.size();

//对应回去,标签是起始点,则保持不变

HLabel\* splitedLabel = new\_label\_bucket[nodeId];

IndexType eS = ep.start\_;

IndexType eE = ep.end\_;

std::*cout*<<"thread waking"<<std::endl;

//int returncode = QObject::connect(this,SIGNAL(testSignal(QString)), Global\_Window,SLOT(logText(QString)),Qt::QueuedConnection

//QMetaObject::invokeMethod( Global\_Window ,"logText",Qt::QueuedConnection ,Q\_ARG(QString ,QString("12345"))/\*,Q\_ARG(int,0)\*/);

/\*bool returncode;

QMetaObject::invokeMethod( Global\_Window ,"logText",Qt::QueuedConnection ,Q\_RETURN\_ARG(bool ,returncode) ,Q\_ARG(char\* ,"12345"),Q\_ARG(int,0) );\*/

Logger<<"边的起点为"<<eS<<"终点为"<<eE<<endl;

emit writeSignal(QString("begin: edge start: ")+ QString("%1").arg(eS)+QString(" edge end: ")+QString("%1").arg(eE) ,3); //释放写信号,写到主窗口的输出框中

QMutex mutex;

mutex.lock();

mWaitcond.wait(&mutex);

mutex.unlock();

std::*cout*<<"thread waked"<<std::endl;

IndexType recordS = 0;

IndexType recordE = 0;

IndexType vtxBSize = splitedLabel->vertex\_bucket.size();

for (auto iter = splitedLabel->vertex\_bucket.begin(); iter != splitedLabel->vertex\_bucket.end(); iter ++)

{

IndexType prev\_id = iter->second->prev\_corr->label\_parent[gLevel]->label\_id; //

if (prev\_id == eS)

{

recordS ++;

}else if(prev\_id == eE)

{

recordE ++;

}

}

ScalarType ration = (ScalarType)(recordS + recordE)/vtxBSize;

emit writeSignal(QString("recordS: ")+ QString("%1").arg(recordS)+QString(" recordE: ")+QString("%1").arg(recordE)+QString(" vtxBSize: ")+QString("%1").arg(vtxBSize) ,3); //释放写信号,写到主窗口的输出框中

//若分裂出来的点个数有一个数据很少,则该边不做裂变

if ( recordE < 5 || recordS < 5 )

{

Logger<<"边界点太靠近,不需要分裂.\n";

emit writeSignal(QString("edge point is not enough, do not split.\n"),1); //释放写信号,写到主窗口的输出框中

continue;

}

if (ration < 0.1) //0.2

{

emit writeSignal(QString("Unmark ratio too high,do not split now.\n"),1);

Logger<<"Unmark点比值太大,暂时不分裂.\n";

continue;

}

//遍历nodeId 上相连接的边

pair<VertexIterator, VertexIterator> vi = boost::vertices(\*new\_graph);

VertexIterator nodeIter = (vi.first + nodeId);

VertexDescriptor nodeDesc = \*nodeIter;

//节点对应的所有出边

pair<OutEdgeIterator,OutEdgeIterator> nodeEiter = boost::out\_edges(nodeDesc,\*new\_graph);

map<IndexType,map<IndexType,HVertex\*> > recordColapseEdges;

set<GraphEdgeProperty> collapseEdges;

OutEdgeIterator oit,nextIt;

oit = nodeEiter.first;

for (nextIt = oit; oit != nodeEiter.second; oit = nextIt )

{

++nextIt;

EdgeDescriptor nextEdgeD = \*oit;

GraphEdgeProperty& nextEP = (\*new\_graph)[nextEdgeD];

collapseEdges.insert(nextEP);

boost::remove\_edge(\*oit,\*new\_graph);//删除这条边

}

//增加一个节点

IndexType nSize = boost::num\_vertices(\*new\_graph);

GraphVertexProperty vp(nSize,-1,-1);

boost::add\_vertex(vp,\*new\_graph);

//更新分割块信息,新增加的Label标号为nSize. 被分裂的点为nodeId

IndexType new\_label = nSize;

new\_label\_bucket.*push\_back*((HLabel\*)0 );

HLabel\* new\_label\_space = dp\_.allocator\_.allocate<HLabel>();

HLabel\* new\_label\_obj = new (new\_label\_space)HLabel;

new\_label\_bucket[new\_label] = new\_label\_obj;

new\_label\_bucket[new\_label]->label\_id = new\_label;

new\_label\_bucket[new\_label]->frame\_parent = &dp\_.hier\_componets\_[tgFrame];

// //对应回去,标签是起始点,则保持不变

map<IndexType,HVertex\*> unMakePs;

for (auto iter = splitedLabel->vertex\_bucket.begin(); iter != splitedLabel->vertex\_bucket.end(); )

{

IndexType prev\_id = iter->second->prev\_corr->label\_parent[gLevel]->label\_id; //得不到最新的label\_parent地址.

IndexType vtx\_id = iter->first;

if (prev\_id == eS)

{

//iter->second->label\_parent.push\_back( new\_label\_bucket[nodeId] );//每次赋了标签之后,都会添加一个地址,这样会导致父节点很多,而我们只需要最终的父节点地址

iter->second->label\_parent.resize(tgGraSize + 1);

iter->second->label\_parent[tgGraSize] = new\_label\_bucket[nodeId];

++iter;

}else if(prev\_id == eE)

{

//iter->second->label\_parent.push\_back(new\_label\_bucket[new\_label] );

iter->second->label\_parent.resize(tgGraSize + 1);

iter->second->label\_parent[tgGraSize] = new\_label\_bucket[new\_label] ;

new\_label\_bucket[new\_label]->vertex\_bucket.insert(\*iter);

dp\_.hier\_componets\_[tgFrame].label\_of\_vtx[vtx\_id ] = new\_label;

iter = splitedLabel->vertex\_bucket.erase(iter);

}else//一些待确定label的点

{

unMakePs.insert(\*iter);

iter = splitedLabel->vertex\_bucket.erase(iter);

}

}

//用随机取点产生的最小距离来判断不确定点属于哪个类.unmark 要么属于nodeid 要么属于new\_label

determinateUnmarkPoints(tgFrame,unMakePs,new\_label\_bucket,nodeId,new\_label,tgGraSize);

//对这两个点进行加边操作,

//node<--->new\_node

GraphEdgeProperty newEP;

newEP.start\_ = nodeId;

newEP.end\_ = nSize;

newEP.index = newGraphEdgeSize;

IndexType edgeKey = frame\_index\_to\_key(newEP.start\_,newEP.end\_);

//还没对index赋值

//

// newEP.edgePoints[edgeKey].insert(edgeCorrNextVtx.begin(),edgeCorrNextVtx.end());

map<IndexType, map<IndexType ,HVertex\*> > nedgePoints;

getEdgePoints( new\_label\_bucket[nodeId] , new\_label\_bucket[new\_label] ,tgFrame,nedgePoints);

newEP.edgePoints[edgeKey].insert(nedgePoints[edgeKey].begin(), nedgePoints[edgeKey].end());

boost::add\_edge(nodeId,nSize,newEP,\*new\_graph); //nodeId: the node split. nSize: nSize: the node added

////the callapsed edged may be out of work ,it is right for us to iterate the left node to build the edges

for (auto iter = collapseEdges.begin(); iter != collapseEdges.end(); iter ++)

{

GraphEdgeProperty glueEdge;

glueEdge = (\*iter);

IndexType sEdgeId = glueEdge.start\_; //the start point of the edge ,this id if smaller than the eEdgeId

IndexType eEdgeId = glueEdge.end\_; //the end point of the edge

map<IndexType,HVertex\*> edgePoints;

auto bIter = glueEdge.edgePoints.begin();

edgePoints.insert(bIter->second.begin(),bIter->second.end() );

map<IndexType,HVertex\*> startVtx = new\_label\_bucket[sEdgeId]->vertex\_bucket;

map<IndexType,HVertex\*> endVtx = new\_label\_bucket[eEdgeId]->vertex\_bucket;

map<IndexType,HVertex\*> nodeVtx = new\_label\_bucket[nodeId]->vertex\_bucket;

map<IndexType,HVertex\*> nSizeVtx = new\_label\_bucket[nSize]->vertex\_bucket;

ScalarType minNode = 0.0;

ScalarType minSize = 0.0;

if (sEdgeId < nodeId) //nodeid为终点

{

minDistBeTwoParts(tgFrame,startVtx,nodeVtx,minNode);

minDistBeTwoParts(tgFrame,startVtx,nSizeVtx,minSize);

if( minNode < 0.04 && minSize <0.04)

{

glueEdge.end\_ = nodeId;

IndexType eKey1 = frame\_index\_to\_key(glueEdge.start\_,glueEdge.end\_);

glueEdge.edgePoints[eKey1] = edgePoints;

boost::add\_edge(glueEdge.start\_,glueEdge.end\_,glueEdge,\*new\_graph);

IndexType newGraphEdgeSize = new\_graph->m\_edges.size(); //为了给新增加的边添加序号

GraphEdgeProperty addglueEdge;

addglueEdge.start\_ = glueEdge.start\_;

addglueEdge.end\_ = nSize;

addglueEdge.index = newGraphEdgeSize ;

IndexType eKey2 = frame\_index\_to\_key(addglueEdge.start\_,addglueEdge.end\_);

addglueEdge.edgePoints[eKey2] = edgePoints;

boost::add\_edge(addglueEdge.start\_,addglueEdge.end\_,addglueEdge,\*new\_graph);

continue;

}

else if (minNode < minSize)

{

glueEdge.end\_ = nodeId;

}else

{

glueEdge.end\_ = nSize;

}

}else//nodeid为起点

{

minDistBeTwoParts(tgFrame,endVtx,nodeVtx,minNode);

minDistBeTwoParts(tgFrame,endVtx,nSizeVtx,minSize);

if( minNode < 0.04 && minSize <0.04)

{

glueEdge.start\_ = nodeId;

glueEdge.end\_ = eEdgeId;

IndexType eKey1 = frame\_index\_to\_key(glueEdge.start\_,glueEdge.end\_);

glueEdge.edgePoints[eKey1] = edgePoints;

boost::add\_edge(glueEdge.start\_,glueEdge.end\_,glueEdge,\*new\_graph);

IndexType newGraphEdgeSize = new\_graph->m\_edges.size(); //为了给新增加的边添加序号

GraphEdgeProperty addglueEdge;

addglueEdge.start\_ = eEdgeId;

addglueEdge.end\_ = nSize;

addglueEdge.index = newGraphEdgeSize ;

IndexType eKey2 = frame\_index\_to\_key(addglueEdge.start\_,addglueEdge.end\_);

addglueEdge.edgePoints[eKey2] = edgePoints;

boost::add\_edge(addglueEdge.start\_,addglueEdge.end\_,addglueEdge,\*new\_graph);

continue;

}

else if (minNode < minSize)

{

glueEdge.start\_ = nodeId;

glueEdge.end\_ = eEdgeId;

}else

{

glueEdge.start\_ = eEdgeId;

glueEdge.end\_ = nSize;

}

}

IndexType eKey = frame\_index\_to\_key(glueEdge.start\_,glueEdge.end\_);

glueEdge.edgePoints[eKey] = edgePoints;

boost::add\_edge(glueEdge.start\_,glueEdge.end\_,glueEdge,\*new\_graph);

}//遍历collapse的边

//这里测试 ，直接替换第一层的label\_bucket

dp\_.hier\_componets\_[tgFrame].hier\_label\_bucket[0] = new\_label\_bucket;

map<IndexType,IndexType> labelIndex;

IndexType kk=0;

for (auto iter = new\_label\_bucket.*begin*(); iter != new\_label\_bucket.*end*(); ++ iter,++kk)

{

IndexType label = (\*iter)->label\_id;

labelIndex[label] = kk;

}

dp\_.hier\_componets\_[tgFrame].hier\_label\_vtxBucket\_index[0] = labelIndex;

dp\_.hier\_componets\_[tgFrame].hier\_graph[0] = new\_graph;

// dp\_.init\_labeles\_graph\_hier2(0); //in fact this function is build graph based on space disatance ,therefore ,it may not be the reality graph

dp\_.init\_node\_link(0);

dp\_.changedDepthAndDispaly(0);

QMetaObject::invokeMethod( Global\_Window->getCanvas() ,"updateGL",Qt::QueuedConnection);

emit writeSignal(QString("target frame")+QString("%1").arg(tgFrame)+QString("frame iter one end"),2); //释放写信号,写到主窗口的输出框中

//Global\_Window->getCanvas()->updateGL();

}//遍历每条边,每条边都会使得new\_graph增加一个新的节点

emit writeSignal(QString("target frame")+QString("%1").arg(tgFrame)+QString("all iter end"),2); //释放写信号,写到主窗口的输出框中

checkPsNewLabelParentPtr(new\_label\_bucket,labParentsize);//next dirction

map<IndexType,IndexType> labelIndex;

IndexType kk=0;

for (auto iter = new\_label\_bucket.*begin*(); iter != new\_label\_bucket.*end*(); ++ iter,++kk)

{

IndexType label = (\*iter)->label\_id;

labelIndex[label] = kk;

}

dp\_.hier\_componets\_[tgFrame].hier\_label\_bucket.push\_back(new\_label\_bucket);

dp\_.hier\_componets\_[tgFrame].hier\_graph.push\_back(new\_graph);//保存最新的graph

dp\_.hier\_componets\_[tgFrame].hier\_label\_vtxBucket\_index.push\_back(labelIndex);

//main\_window\_->getCanvas()->updateGL();

Logger<<" Start next split.\n";

Logger<<" .......\n";

}

void PropagateThread::split\_twoAjacent\_graph\_prev( DualwayPropagation& dp ,IndexType srFrame, IndexType tgFrame)

{

//获取需要更新的graph

Logger<<" .......\n";

Logger<<" Start prev split.\n";

IndexType srGraphSize = dp\_.hier\_componets\_[srFrame].hier\_graph.size();

IndexType tgGraphSize = dp\_.hier\_componets\_[tgFrame].hier\_graph.size();

assert(srGraphSize > 0 && tgGraphSize > 0);

LabelsGraph\* oriSpGra = dp\_.hier\_componets\_[srFrame].hier\_graph[srGraphSize - 1];

LabelsGraph\* shouldSplitGraph = new LabelsGraph(\*oriSpGra);

*vector*<HLabel\* > new\_label\_bucket;

copyLabelBucket(new\_label\_bucket,dp\_.hier\_componets\_[srFrame].hier\_label\_bucket[srGraphSize - 1] );

//获取指导分割的graph

LabelsGraph\* guideSplitGraph;

IndexType graphLevel = 0;

if (tgGraphSize > 1)

{

graphLevel = tgGraphSize - 2;

}

graphLevel = tgGraphSize -1 ;

guideSplitGraph = dp\_.hier\_componets\_[tgFrame].hier\_graph[graphLevel];

// guideSplitGraph的每条边引导一次分割

pair<EdgeIterator,EdgeIterator> ei = boost::edges(\*guideSplitGraph);

emit writeSignal(QString("target frame")+QString("%1").arg(tgFrame)+QString("begin"),1);

for (EdgeIterator eit = ei.first; eit != ei.second; ++eit)

{

EdgeDescriptor ed = \*eit;

GraphEdgeProperty& ep = (\*guideSplitGraph)[ed];

map<IndexType,HVertex\*> edgePoints;

auto ePsIt = ep.edgePoints.begin();

edgePoints.insert(ePsIt->second.begin(),ePsIt->second.end() );

if (edgePoints.size() < 1)

{

Logger<<"边上的顶点数太少,无法分裂.\n";

continue;

}

map<IndexType,HVertex\*> edgeCorrPrevVtx;

IndexType nSize = boost::num\_vertices(\*shouldSplitGraph);

IndexType newGraphEdgeSize = shouldSplitGraph->m\_edges.size(); //为了给新增加的边添加序号

bool isSplit = true;

IndexType edgePsCorNode = 0;

isSplit = checkPrevLabelBucket(edgePoints,edgeCorrPrevVtx,edgePsCorNode);

if ( edgePsCorNode < 0 || edgePsCorNode > nSize - 1)

{

Logger<<"边界找到的块越界!.\n";

break;

}

if (!isSplit)

{

Logger<<"边界点指向了两个块,不分割.\n";

continue;

}

HLabel\* splitedLabel = new\_label\_bucket[edgePsCorNode]; //可能需要分裂的块

IndexType curEdgeStart = ep.start\_;

IndexType curEdgeEnd = ep.end\_;

IndexType strCorPsSzie = 0;

IndexType endCorPsSize = 0;

std::*cout*<<"thread waking"<<std::endl;

Logger<<"边的起点为"<<curEdgeStart<<"终点为"<<curEdgeEnd<<endl;

emit writeSignal(QString("begin: edge start: ")+ QString("%1").arg(curEdgeStart)+QString(" edge end: ")+QString("%1").arg(curEdgeEnd) ,3); //释放写信号,写到主窗口的输出框中

QMutex mutex;

mutex.lock();

mWaitcond.wait(&mutex);

mutex.unlock();

std::*cout*<<"thread waked"<<std::endl;

//分类前做一个简单的预判断

for (auto iter = splitedLabel->vertex\_bucket.begin(); iter != splitedLabel->vertex\_bucket.end(); iter ++)

{

if (iter->second->next\_corr->label\_parent[graphLevel] != *NULL*)

{

IndexType nextVtx\_label = iter->second->next\_corr->label\_parent[graphLevel]->label\_id;

// IndexType nextCor\_id = iter->second->next\_corr->vtx\_id;

//

// IndexType nextVtx\_label = hier\_componets\_[tgFrame].label\_of\_vtx[nextCor\_id];

if (nextVtx\_label == curEdgeStart)

{

strCorPsSzie ++;

}else if(nextVtx\_label == curEdgeEnd)

{

endCorPsSize ++;

}

}

}

IndexType vtxBSize = splitedLabel->vertex\_bucket.size();

ScalarType ration = (ScalarType)(strCorPsSzie + endCorPsSize)/vtxBSize;

emit writeSignal(QString("recordS: ")+ QString("%1").arg(strCorPsSzie)+QString(" recordE: ")+QString("%1").arg(endCorPsSize) +QString(" vtxBSize: ")+QString("%1").arg(vtxBSize),3); //释放写信号,写到主窗口的输出框中

if ( strCorPsSzie < 5 || endCorPsSize < 5)

{

Logger<<"边界点靠近边界(对应点个数),不需要分裂.\n";

emit writeSignal(QString("edge point is not enough, do not split.\n"),1); //释放写信号,写到主窗口的输出框中

continue;

}

if ( ration < 0.05) //0.2

{

Logger<<"四两拨千斤?算了吧!.\n";

emit writeSignal(QString("Unmark ratio too high,do not split now.\n"),1);

continue;

}

//用eit 边开始分裂

pair<VertexIterator, VertexIterator> vi = boost::vertices(\*shouldSplitGraph);

VertexIterator curNodeIter = (vi.first + edgePsCorNode);

VertexDescriptor curNodeDesc = \*curNodeIter;

//节点对应的所有出边

pair<OutEdgeIterator,OutEdgeIterator> nodeEiter = boost::out\_edges(curNodeDesc,\*shouldSplitGraph);

map<IndexType,map<IndexType,HVertex\*> > recordColapseEdges;

set<GraphEdgeProperty> collapseEdges;

OutEdgeIterator oit,nextIt;

oit = nodeEiter.first;

for (nextIt = oit; oit != nodeEiter.second; oit = nextIt )

{

++nextIt;

EdgeDescriptor nextEdgeD = \*oit;

GraphEdgeProperty& nextEP = (\*shouldSplitGraph)[nextEdgeD];

collapseEdges.insert(nextEP);

boost::remove\_edge(\*oit,\*shouldSplitGraph);//删除这条边

}

//增加一个节点

GraphVertexProperty vp(nSize,-1,-1);

boost::add\_vertex(vp,\*shouldSplitGraph);

//更新分割块信息,新增加的Label标号为nSize. 被分裂的点为edgePsCorNode

IndexType new\_label = nSize;

new\_label\_bucket.push\_back((HLabel\*)0 );

HLabel\* new\_label\_space = dp\_.allocator\_.allocate<HLabel>();

HLabel\* new\_label\_obj = new (new\_label\_space)HLabel;

new\_label\_bucket[new\_label] = new\_label\_obj;

new\_label\_bucket[new\_label]->label\_id = new\_label;

new\_label\_bucket[new\_label]->frame\_parent = &dp\_.hier\_componets\_[tgFrame];

//开始分裂

map<IndexType,HVertex\*> unMakePs;

for (auto iter = splitedLabel->vertex\_bucket.begin(); iter != splitedLabel->vertex\_bucket.end(); )

{

if (iter->second->next\_corr->label\_parent[graphLevel])

{

IndexType nextVtx\_label = iter->second->next\_corr->label\_parent[graphLevel]->label\_id; //得不到最新的label\_parent地址.

// IndexType nextCor\_id = iter->second->next\_corr->vtx\_id;

// IndexType nextVtx\_label = hier\_componets\_[tgFrame].label\_of\_vtx[nextCor\_id];

IndexType vtx\_id = iter->first;

if (nextVtx\_label == curEdgeStart)

{

//iter->second->label\_parent.push\_back(new\_label\_bucket[edgePsCorNode] );

iter->second->label\_parent.resize(srGraphSize + 1);

iter->second->label\_parent[srGraphSize] = new\_label\_bucket[edgePsCorNode];

++iter;

}else if(nextVtx\_label == curEdgeEnd)

{

iter->second->label\_parent.resize(srGraphSize + 1);

iter->second->label\_parent[srGraphSize] = new\_label\_bucket[new\_label];

//iter->second->label\_parent.push\_back(new\_label\_bucket[new\_label] );

new\_label\_bucket[new\_label]->vertex\_bucket.insert(\*iter);

dp\_.hier\_componets\_[srFrame].label\_of\_vtx[vtx\_id ] = new\_label;

iter = splitedLabel->vertex\_bucket.erase(iter);

}else//一些待确定label的点

{

unMakePs.insert(\*iter);

iter = splitedLabel->vertex\_bucket.erase(iter);

}

}

} //遍历需要分裂块的每个点

//用随机取点产生的最小距离来判断不确定点属于哪个类.unmark 要么属于nodeid 要么属于new\_label

determinateUnmarkPoints(srFrame,unMakePs,new\_label\_bucket,edgePsCorNode,new\_label,srGraphSize);

//对这两个点进行加边操作,

//node<--->new\_node

GraphEdgeProperty newEP;

newEP.start\_ = edgePsCorNode;

newEP.end\_ = nSize;

newEP.index = newGraphEdgeSize;

IndexType edgeKey = frame\_index\_to\_key(newEP.start\_,newEP.end\_);

newEP.edgePoints[edgeKey].insert(edgeCorrPrevVtx.begin(),edgeCorrPrevVtx.end());

boost::add\_edge(edgePsCorNode,nSize,newEP,\*shouldSplitGraph);

//断定查找两个节点与其它节点进行连边操作只会出现 recordColapseEdges.size()次数.

for (auto iter = collapseEdges.begin(); iter != collapseEdges.end(); iter ++)

{

GraphEdgeProperty glueEdge;

glueEdge = (\*iter);

IndexType sEdgeId = glueEdge.start\_;

IndexType eEdgeId = glueEdge.end\_;

map<IndexType,HVertex\*> edgePoints;

auto bIter = glueEdge.edgePoints.begin();

edgePoints.insert(bIter->second.begin(),bIter->second.end() );

map<IndexType,HVertex\*> startVtx = new\_label\_bucket[sEdgeId]->vertex\_bucket;

map<IndexType,HVertex\*> endVtx = new\_label\_bucket[eEdgeId]->vertex\_bucket;

map<IndexType,HVertex\*> nodeVtx = new\_label\_bucket[edgePsCorNode]->vertex\_bucket;

map<IndexType,HVertex\*> nSizeVtx = new\_label\_bucket[nSize]->vertex\_bucket;

ScalarType minNode = 0.0;

ScalarType minSize = 0.0;

if (sEdgeId < edgePsCorNode) //nodeid为终点

{

minDistBeTwoParts(srFrame,startVtx,nodeVtx,minNode);

minDistBeTwoParts(srFrame,startVtx,nSizeVtx,minSize);

if( minNode < 0.04 && minSize <0.04)

{

glueEdge.end\_ = edgePsCorNode;

IndexType eKey1 = frame\_index\_to\_key(glueEdge.start\_,glueEdge.end\_);

glueEdge.edgePoints[eKey1] = edgePoints;

boost::add\_edge(glueEdge.start\_,glueEdge.end\_,glueEdge,\*shouldSplitGraph);

IndexType newGraphEdgeSize = shouldSplitGraph->m\_edges.size(); //为了给新增加的边添加序号

GraphEdgeProperty addglueEdge;

addglueEdge.start\_ = glueEdge.start\_;

addglueEdge.end\_ = nSize;

addglueEdge.index = newGraphEdgeSize ;

IndexType eKey2 = frame\_index\_to\_key(addglueEdge.start\_,addglueEdge.end\_);

addglueEdge.edgePoints[eKey2] = edgePoints;

boost::add\_edge(addglueEdge.start\_,addglueEdge.end\_,addglueEdge,\*shouldSplitGraph);

continue;

}

else if (minNode < minSize)

{

glueEdge.end\_ = edgePsCorNode;

}else

{

glueEdge.end\_ = nSize;

}

}else//nodeid为起点

{

minDistBeTwoParts(srFrame,endVtx,nodeVtx,minNode);

minDistBeTwoParts(srFrame,endVtx,nSizeVtx,minSize);

if( minNode < 0.04 && minSize <0.04)

{

glueEdge.start\_ =edgePsCorNode;

glueEdge.end\_ = eEdgeId;

IndexType eKey1 = frame\_index\_to\_key(glueEdge.start\_,glueEdge.end\_);

glueEdge.edgePoints[eKey1] = edgePoints;

boost::add\_edge(glueEdge.start\_,glueEdge.end\_,glueEdge,\*shouldSplitGraph);

IndexType newGraphEdgeSize = shouldSplitGraph->m\_edges.size(); //为了给新增加的边添加序号

GraphEdgeProperty addglueEdge;

addglueEdge.start\_ = eEdgeId;

addglueEdge.end\_ = nSize;

addglueEdge.index = newGraphEdgeSize ;

IndexType eKey2 = frame\_index\_to\_key(addglueEdge.start\_,addglueEdge.end\_);

addglueEdge.edgePoints[eKey2] = edgePoints;

boost::add\_edge(addglueEdge.start\_,addglueEdge.end\_,addglueEdge,\*shouldSplitGraph);

continue;

}

else if (minNode < minSize)

{

glueEdge.start\_ = edgePsCorNode;

glueEdge.end\_ = eEdgeId;

}else

{

glueEdge.start\_ = eEdgeId;

glueEdge.end\_ = nSize;

}

}

IndexType eKey = frame\_index\_to\_key(glueEdge.start\_,glueEdge.end\_);

glueEdge.edgePoints[eKey] = edgePoints;

boost::add\_edge(glueEdge.start\_,glueEdge.end\_,glueEdge,\*shouldSplitGraph);

}//遍历collapse的边

dp\_.hier\_componets\_[srFrame].hier\_label\_bucket[0] = new\_label\_bucket;

map<IndexType,IndexType> labelIndex;

IndexType kk=0;

for (auto iter = new\_label\_bucket.begin(); iter != new\_label\_bucket.end(); ++ iter,++kk)

{

IndexType label = (\*iter)->label\_id;

labelIndex[label] = kk;

}

dp\_.hier\_componets\_[srFrame].hier\_label\_vtxBucket\_index[0] = labelIndex;

dp\_.hier\_componets\_[srFrame].hier\_graph[0] = shouldSplitGraph;

dp\_.init\_node\_link(0);

dp\_.changedDepthAndDispaly(0);

QMetaObject::invokeMethod( Global\_Window->getCanvas() ,"updateGL",Qt::QueuedConnection);

emit writeSignal(QString("target frame")+QString("%1").arg(tgFrame)+QString("frame iter one end"),2); //释放写信号,写到主窗口的输出框中

//Global\_Window->getCanvas()->updateGL();

} //遍历引导分割图的每条边

emit writeSignal(QString("target frame")+QString("%1").arg(tgFrame)+QString("all iter end"),2);

checkPsNewLabelParentPtr(new\_label\_bucket,srGraphSize + 1);

map<IndexType,IndexType> labelIndex;

IndexType kk=0;

for (auto iter = new\_label\_bucket.begin(); iter != new\_label\_bucket.end(); ++ iter,++kk)

{

IndexType label = (\*iter)->label\_id;

labelIndex[label] = kk;

}

dp\_.hier\_componets\_[srFrame].hier\_label\_bucket.push\_back(new\_label\_bucket);

dp\_.hier\_componets\_[srFrame].hier\_graph.push\_back(shouldSplitGraph);//保存最新的graph

dp\_.hier\_componets\_[tgFrame].hier\_label\_vtxBucket\_index.push\_back(labelIndex);

Logger<<" End prev split.\n";

Logger<<" .......\n";

}