

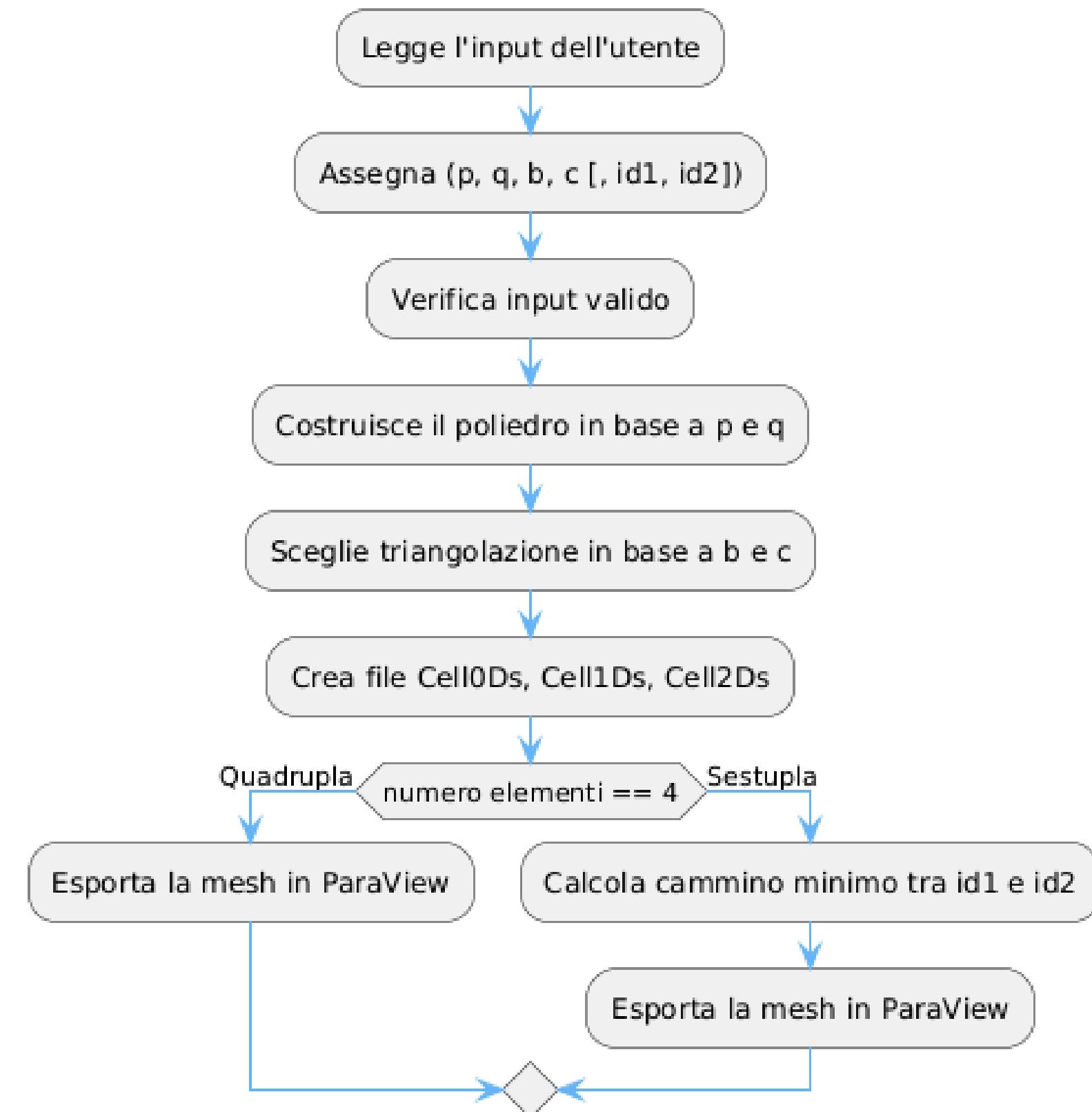
PROGETTO P C S

Meregalli Emma s310668
Montano Giorgia s310173
Salvetti Chiara s310925



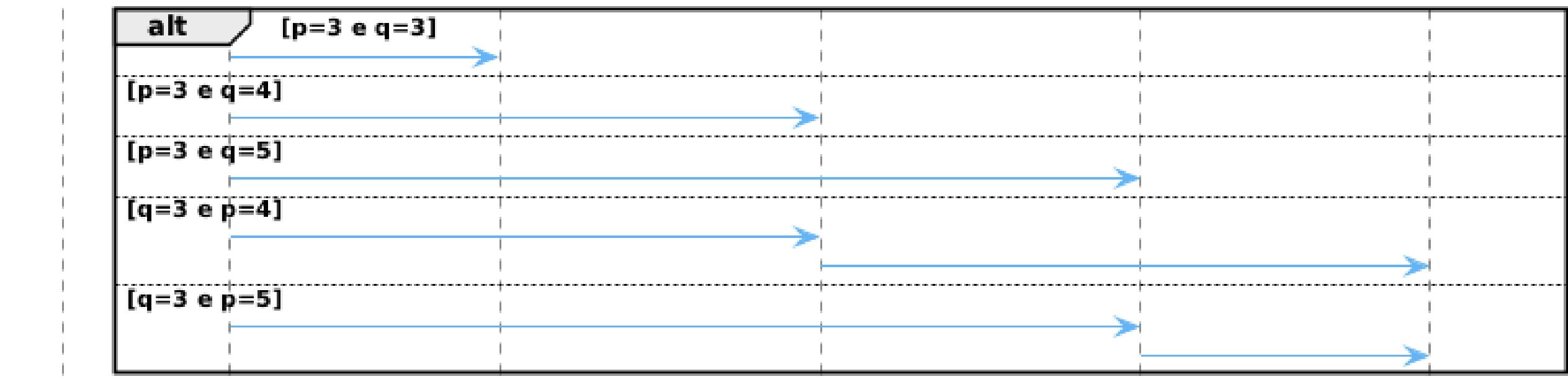
Politecnico
di Torino

main.cpp





Utente



Utente



PolyhedralMesh.hpp & Utils.hpp

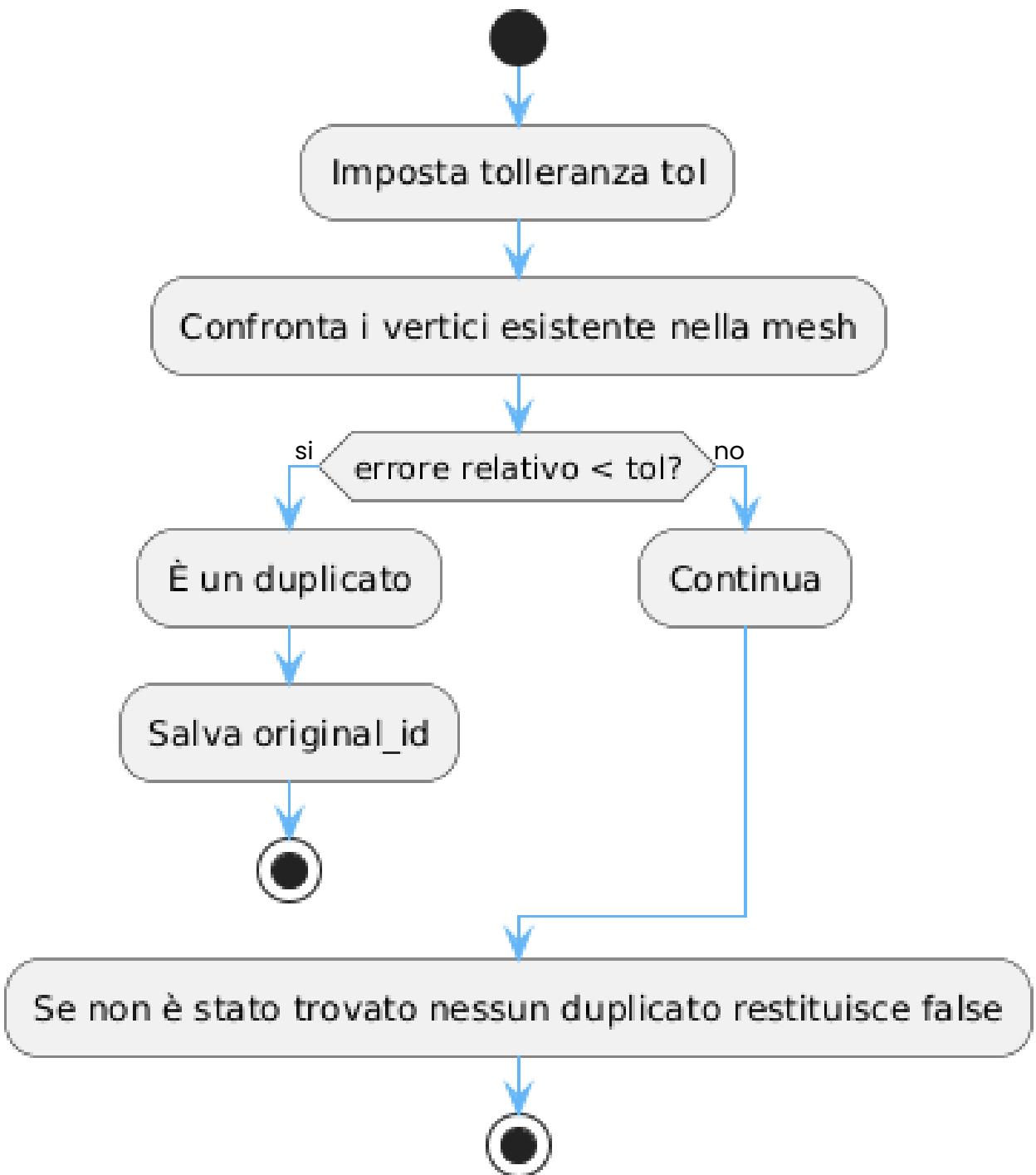
C PolyhedralMesh

- unsigned int NumCell0Ds
- vector<unsigned int> Cell0DsId
- MatrixXd Cell0DsCoordinates
- unsigned int NumCell1Ds
- vector<unsigned int> Cell1DsId
- MatrixXi Cell1DsExtrema
- unsigned int NumCell2Ds
- vector<unsigned int> Cell2DsId
- vector<unsigned int> Cell2DsNumVertices
- vector<unsigned int> Cell2DsNumEdges
- vector<vector<unsigned int>> Cell2DsVertices
- vector<vector<unsigned int>> Cell2DsEdges
- unsigned int NumCell3Ds
- vector<unsigned int> Cell3DsId
- vector<unsigned int> Cell3DsVertices
- vector<unsigned int> Cell3DsEdges
- vector<unsigned int> Cell3DsFaces
- vector<unsigned int> Cell0DsShortPath
- vector<unsigned int> Cell1DsShortPath

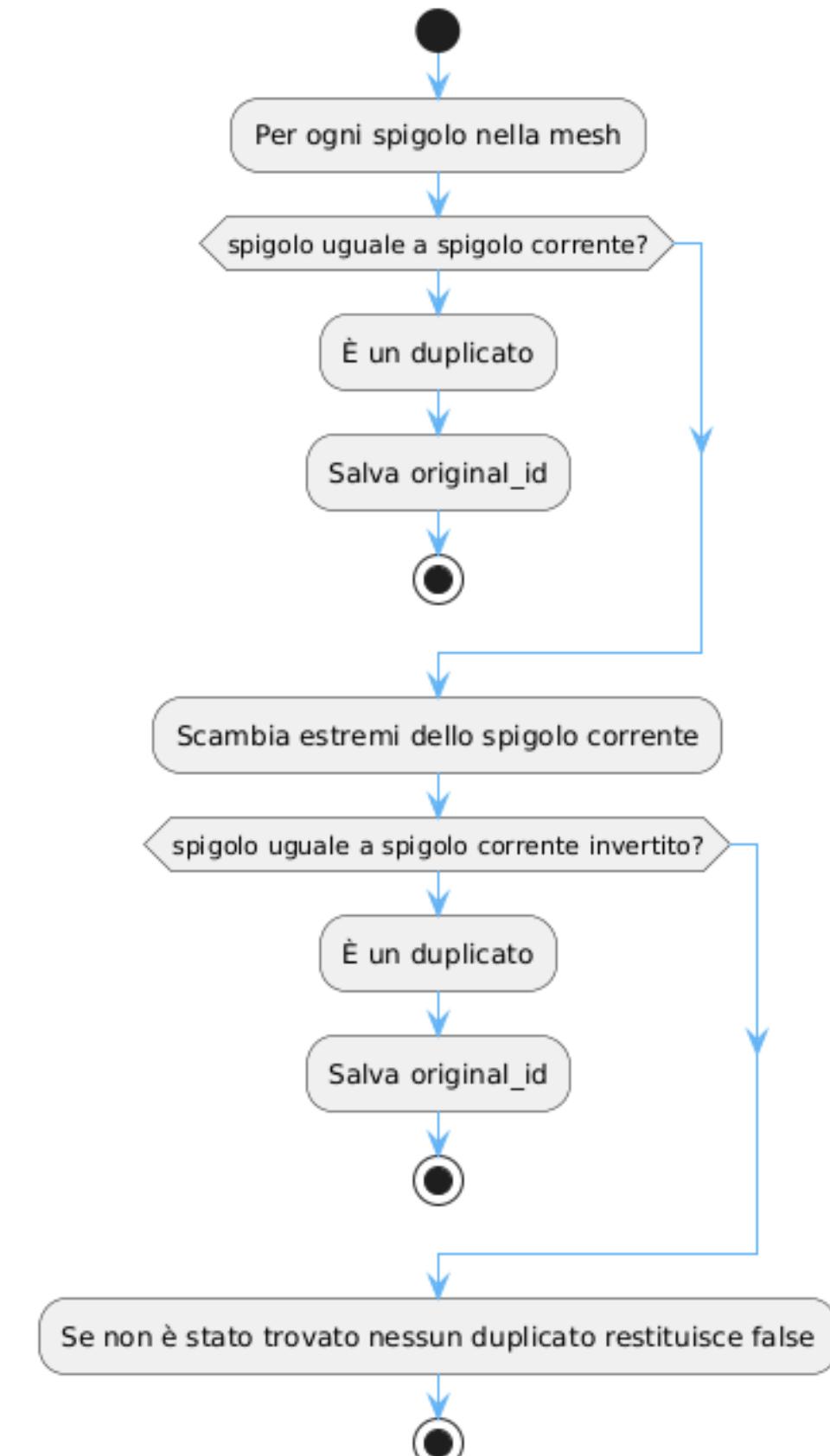
C Utils

- ComputeVEF(unsigned int q, unsigned int b, unsigned int c)
- CreateTxtFiles(const PolyhedralMesh& mesh)
- GenerateDual(const PolyhedralMesh& mesh, PolyhedralMesh& dualMesh)
- ExportTetrahedron(PolyhedralMesh& mesh, PolyhedralMesh& triMesh, const unsigned int& b, const unsigned int& c)
- ExportOctahedron(PolyhedralMesh& mesh, PolyhedralMesh& triMesh, const unsigned int& b, const unsigned int& c)
- ExportIcosahedron(PolyhedralMesh& mesh, PolyhedralMesh& triMesh, const unsigned int& b, const unsigned int& c)
- ExportParaView(PolyhedralMesh& mesh, bool path)
- OrderFaces(const vector<int>& unordered_faces, vector<int>& ordered_faces, const PolyhedralMesh& mesh)
- ShortestPath(PolyhedralMesh& mesh, unsigned int id_vertex_1, unsigned int id_vertex_2)

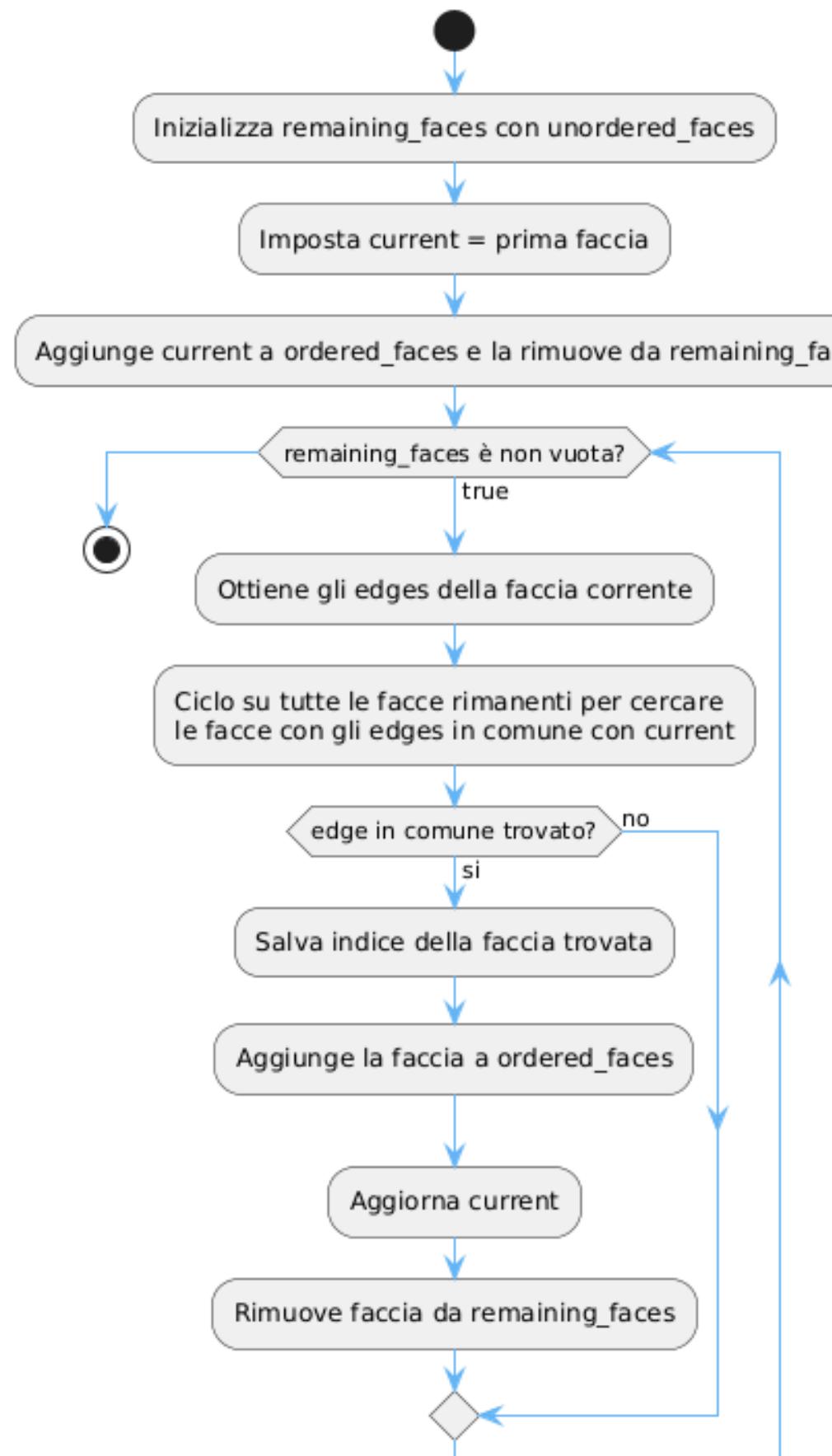
VertexIsDupe



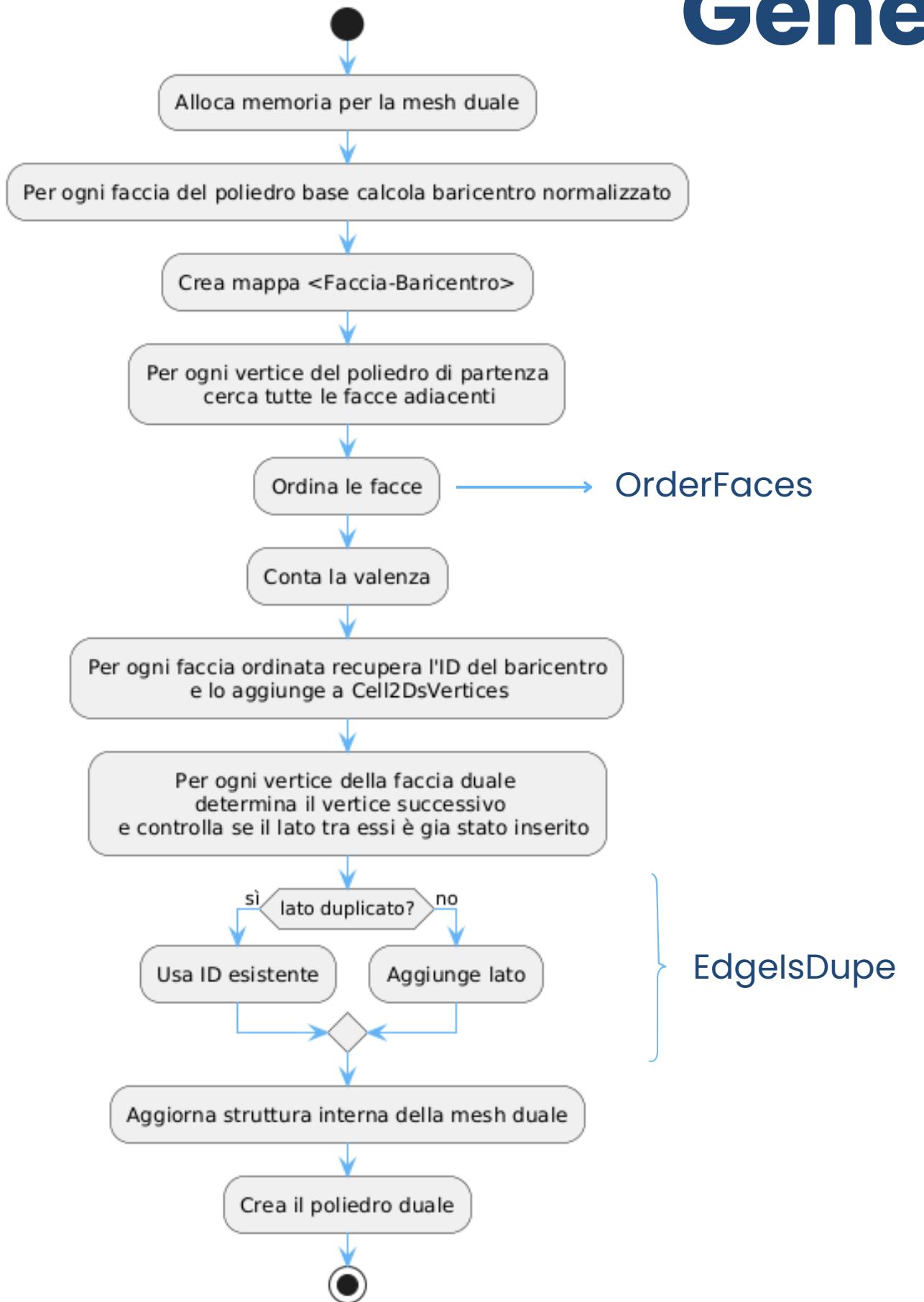
EdgeIsDupe



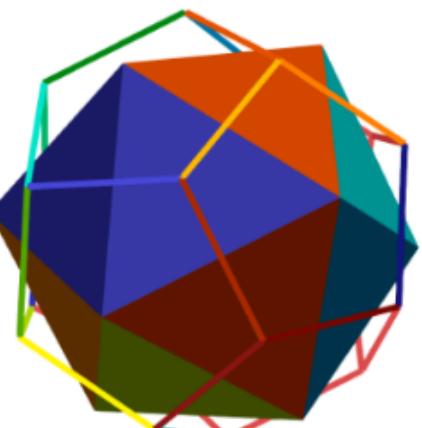
OrderFaces



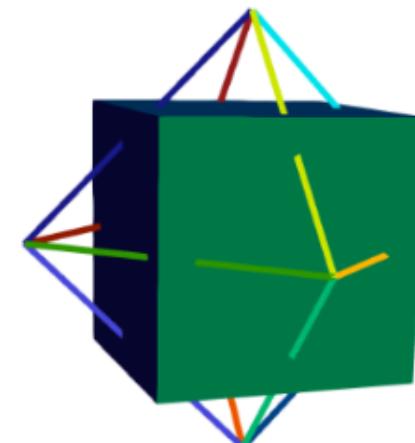
GenerateDual



Tetraedro \longleftrightarrow Tetraedro



Icosaedro \longleftrightarrow Dodecaedro



Cubo \longleftrightarrow Ottaedro

```

//Funzione generatrice del duale
bool GenerateDual(const PolyhedralMesh& baseMesh, PolyhedralMesh& dualMesh) {
    int baricenter_id = 0;
    int face_id = 0;
    int edge_id = 0;

    //Il numero di vertici del duale è uguale al numero di facce del poliedro di partenza
    dualMesh.NumCell0Ds = baseMesh.NumCell2Ds;

    //Il numero dei lati del duale è uguale al numero di lati del poliedro di partenza
    dualMesh.NumCell1Ds = baseMesh.NumCell1Ds;

    //Il numero di facce del duale è uguale al numero di vertici del poliedro di partenza
    dualMesh.NumCell2Ds = baseMesh.NumCell0Ds;
    dualMesh.Cell0DsId.reserve(dualMesh.NumCell0Ds);
    dualMesh.Cell0DsCoordinates = MatrixXd::Zero(3,dualMesh.NumCell0Ds);

    dualMesh.Cell1DsExtrema = MatrixXi::Zero(2, dualMesh.NumCell1Ds);
    dualMesh.Cell1DsId.reserve(dualMesh.NumCell1Ds);

    //Allocazione di memoria
    dualMesh.Cell2DsId.reserve(dualMesh.NumCell2Ds);
    dualMesh.Cell2DsEdges.resize(dualMesh.NumCell2Ds);
    dualMesh.Cell2DsNumEdges.resize(dualMesh.NumCell2Ds);
    dualMesh.Cell2DsVertices.resize(dualMesh.NumCell2Ds);
    dualMesh.Cell2DsNumVertices.resize(dualMesh.NumCell2Ds);
    int duplicate_id = 0;

    //Mappa che associa all'id della faccia l'id del baricentro corrispondente
    map<int, int> Faces_bar;
    for (const auto& id : baseMesh.Cell2DsId) {
        Vector3d baricenter;

        //Salva in 3 vettori le coordinate dei vertici della faccia corrente
        Vector3d Vertex1 = baseMesh.Cell0DsCoordinates.col(baseMesh.Cell2DsVertices[id][0]);
        Vector3d Vertex2 = baseMesh.Cell0DsCoordinates.col(baseMesh.Cell2DsVertices[id][1]);
        Vector3d Vertex3 = baseMesh.Cell0DsCoordinates.col(baseMesh.Cell2DsVertices[id][2]);

        //Calcola le coordinate del baricentro
        baricenter = (1.0 / 3.0) * Vertex1 + (1.0 / 3.0) * Vertex2 + (1.0 / 3.0) * Vertex3;
        baricenter = baricenter / baricenter.norm();
        dualMesh.Cell0DsId.push_back(baricenter_id);
        //Salva le coordinate del baricentro appena trovato in Cell0DsCoordinates del poliedro Duale
        dualMesh.Cell0DsCoordinates(0, id) = baricenter(0);
        dualMesh.Cell0DsCoordinates(1, id) = baricenter(1);
        dualMesh.Cell0DsCoordinates(2, id) = baricenter(2);

        //Associo all'id della faccia l'id del baricentro nella mappa
        Faces_bar[id] = baricenter_id;
        baricenter_id++;
    }
    //Ciclo sui vertici del poliedro di partenza
    for(const auto& vertex_id: baseMesh.Cell0DsId){
        //Inizializza e riempie il vettore che contiene le facce che hanno il vertice in comune
        vector<int> VertexFaces;
        for(const auto& face_id: baseMesh.Cell2DsId){
            for(int j = 0; j < 3; j++){
                if (baseMesh.Cell2DsVertices[face_id][j] == vertex_id){
                    //Se la faccia a cui sono arrivato contiene il vertice la aggiunge al vettore
                    VertexFaces.push_back(face_id);
                    break;
                }
            }
        }
        //Utilizza la funzione OrderFaces per ordinare le facce adiacenti a uno stesso vertice
        //in modo tale che ogni faccia sia seguita da quella a essa ediacente
        vector<int> ordered_faces;
        OrderFaces(VertexFaces, ordered_faces, baseMesh);
    }
}

//La valenza del vertice è pari alla lunghezza del vettore di facce che condividono il vertice corrente
//(le valenze non sono sempre 3 per il generico solido geodetico)
int valence = ordered_faces.size();
vector<unsigned int> New_vertices;

//Associa ad ogni faccia del poliedro di partenza l'id del vertice nel duale corrispondente
for(const auto& VertexFace: ordered_faces)
    New_vertices.push_back(Faces_bar[VertexFace]);

dualMesh.Cell2DsId.push_back(face_id);
dualMesh.Cell2DsVertices[face_id] = New_vertices;
dualMesh.Cell2DsEdges[face_id].resize(valence);

dualMesh.Cell2DsNumVertices[face_id] = valence;
dualMesh.Cell2DsNumEdges[face_id] = valence;

//Crea i lati
//Il vettore di vertici della faccia ha tanti elementi quanti la valenza del vertice
//Ciclo su tutti i vertici della faccia nel duale
for (int k = 0; k < valence; k++) {
    //Prende il vertice corrente della faccia face_id
    int originVertex = dualMesh.Cell2DsVertices[face_id][k];
    int endVertex;
    //identifica il vertice successivo per formare un lato.
    //Se siamo all'ultimo vertice, chiude il ciclo tornando al primo: questo chiude la faccia
    if (k == valence - 1)
        endVertex = dualMesh.Cell2DsVertices[face_id][0];
    else
        endVertex = dualMesh.Cell2DsVertices[face_id][k + 1];
    Vector2i extrema(originVertex,endVertex);
    unsigned int original_id;
    //Controlla se questo lato è già stato inserito per evitare duplicati
    if(!EdgeIsDupe(dualMesh, extrema, original_id)){
        //Se non è ancora stato inserito aggiorna le informazioni relativa al duale
        dualMesh.Cell1DsId.push_back(edge_id);
        dualMesh.Cell1DsExtrema(0, edge_id) = originVertex;
        dualMesh.Cell1DsExtrema(1, edge_id) = endVertex;
        dualMesh.Cell2DsEdges[face_id][k] = edge_id;
        edge_id++;
    }
    else
        dualMesh.Cell2DsEdges[face_id][k] = original_id;
    face_id++;
}

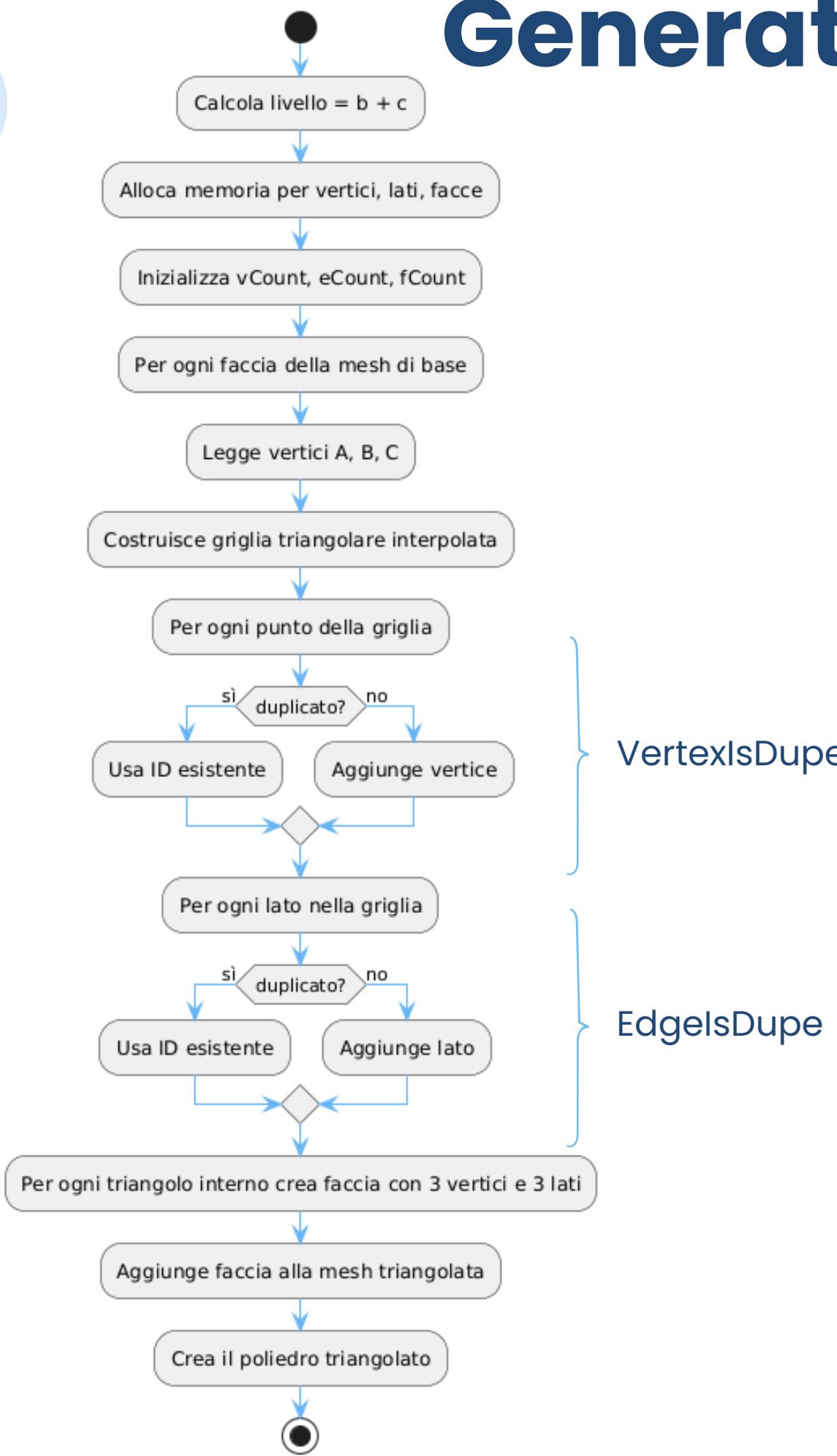
dualMesh.Cell1DsExtrema.conservativeResize(2, dualMesh.NumCell1Ds);
dualMesh.Cell2DsNumVertices.resize(dualMesh.NumCell2Ds);
dualMesh.Cell2DsNumEdges.resize(dualMesh.NumCell2Ds);
dualMesh.Cell2DsVertices.resize(dualMesh.NumCell2Ds);
dualMesh.Cell2DsEdges.resize(dualMesh.NumCell2Ds);

//Genera il poliedro duale
dualMesh.NumCell3Ds++;
dualMesh.Cell3DsId = {0};
dualMesh.Cell3DsVertices = dualMesh.Cell0DsId;
dualMesh.Cell3DsEdges = dualMesh.Cell1DsId;
dualMesh.Cell3DsFaces = dualMesh.Cell2DsId;

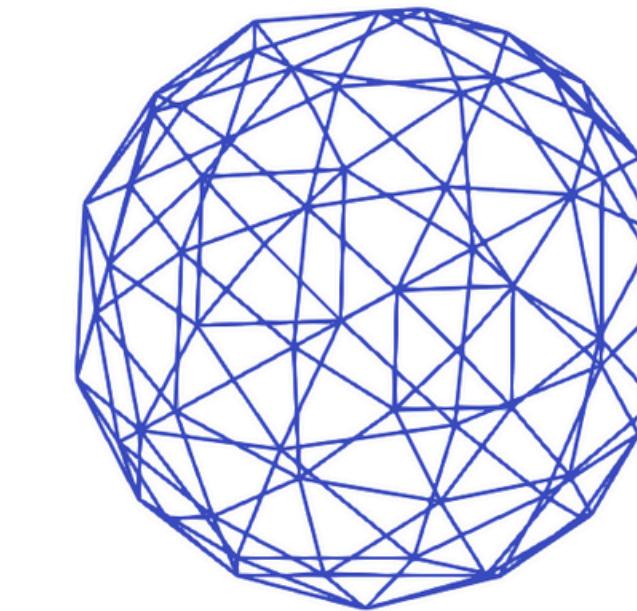
return true;
}

```

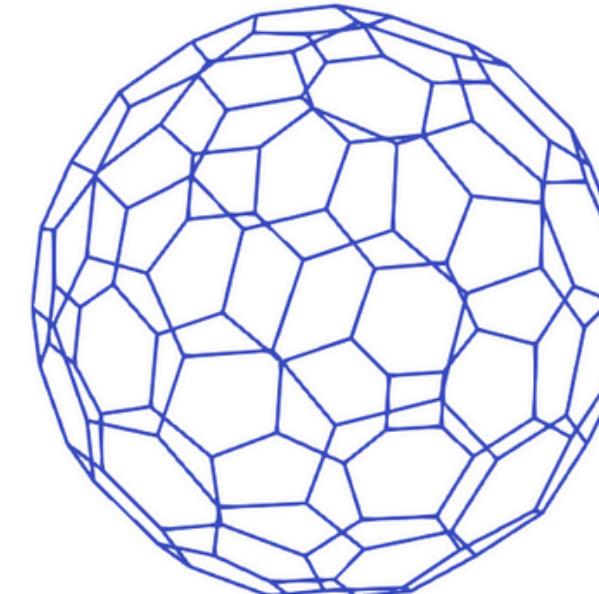
GenerateTriangulatedMesh1



(3,4,4,0)



(4,3,4,0)



```

bool GenerateTriangulatedMesh1(PolyhedralMesh& baseMesh, PolyhedralMesh& triMesh,
    const unsigned int& b, const unsigned int& c, // Parametri della suddivisione
    const Vector3i& triDimensions) // Dimensione di (V,E,F) della mesh triangolata (con duplicati)
{
    unsigned int level = b + c; // Numero di suddivisioni laterali per triangolo

    // Inizializzazione della struttura dati della mesh triangolata

    // Allocazione memoria per vertici (0D)
    triMesh.Cell0DsCoordinates = MatrixXd::Zero(3, triDimensions[0]);
    triMesh.Cell0DsId.reserve(triDimensions[0]);

    // Allocazione memoria per lati (1D)
    triMesh.Cell1DsId.reserve(triDimensions[1]);
    triMesh.Cell1DsExtrema = MatrixXi::Zero(2, triDimensions[1]);

    // Allocazione memoria per facce (2D)
    triMesh.Cell2DsId.reserve(triDimensions[2]);
    triMesh.Cell2DsEdges.reserve(triDimensions[2]);
    triMesh.Cell2DsVertices.reserve(triDimensions[2]);
    for (auto& edgeList : triMesh.Cell2DsEdges){
        edgeList.resize(3); // Ogni faccia ha 3 spigoli
    }
    for (auto& vertList : triMesh.Cell2DsVertices){
        vertList.resize(3); // Ogni faccia ha 3 vertici
    }

    // Contatori per vertici, spigoli e facce
    unsigned int vCount = 0;
    unsigned int eCount = 0;
    unsigned int fCount = 0;
    // Triangolazione per ogni faccia del poliedro iniziale
    for (unsigned int faceIdx = 0; faceIdx < baseMesh.Cell2DsId.size(); faceIdx++){
        const auto& faceVerts = baseMesh.Cell2DsVertices[faceIdx]; // Prende i tre vertici della faccia corrente

        // Coordinate dei 3 vertici del triangolo originale
        Vector3d A = baseMesh.Cell0DsCoordinates.col(faceVerts[0]); // Vertice A
        Vector3d B = baseMesh.Cell0DsCoordinates.col(faceVerts[1]); // Vertice B
        Vector3d C = baseMesh.Cell0DsCoordinates.col(faceVerts[2]); // Vertice C

        vector<vector<unsigned int>> grid; // Griglia di vertici interni alla faccia
        // La griglia ha level = b + c righe e ogni riga i ha i + 1 elementi (forma triangolare)
        // Costruzione della griglia interpolata sulla faccia
        for (unsigned int i = 0; i <= level; i++){
            for (unsigned int j = 0; j <= i; j++) {
                // Interpolo tra from e to per ottenere un punto interno
                Vector3d pos;
                if (i == 0) {
                    pos = A;
                } else {
                    pos = ((double)j / i) * to + ((double)(i - j) / i) * from;
                }
                pos = pos / pos.norm();
                if(!VertexIsDupe(triMesh, pos, original_id)){
                    triMesh.Cell0DsId.push_back(vCount); // Memorizza l'ID del vertice
                    for(unsigned int n = 0; n < 3; n++){
                        triMesh.Cell0DsCoordinates(n,vCount) = pos(n); // Memorizza la posizione
                    }
                    row.push_back(vCount); // Aggiunge l'indice del vertice alla riga corrente
                    vCount++; // Avanza il contatore dei vertici
                }
                else{
                    row.push_back(original_id); // Se il vertice esiste già
                }
            }
            grid.push_back(row); // Aggiunge la riga corrente alla griglia
        }

        // Crea i nuovi lati dati dalla triangolazione e aggiorna la lista dei lati
        unsigned int original_id = 0;

        for(size_t i = 0; i < grid.size(); i++){
            Vector2i extrema;
            for(size_t j = 0; j <= i; j++){
                if(i < grid.size() - 1){
                    // Lato sotto a sinistra
                    extrema[0] = grid[i][j];
                    extrema[1] = grid[i + 1][j];

                    // Aggiorna la lista dei lati, se il vertice non esiste già
                    if(!EdgeIsDupe(triMesh, extrema, original_id)){
                        triMesh.Cell1DsId.push_back(eCount);
                        triMesh.Cell1DsExtrema(0, eCount) = grid[i][j];
                        triMesh.Cell1DsExtrema(1, eCount) = grid[i + 1][j];
                        eCount++;
                    }
                    // Lato sotto a destra
                    extrema[0] = grid[i][j];
                    extrema[1] = grid[i + 1][j+1];
                }
            }
        }
    }

    // Aggiorna la lista dei lati, se il vertice non esiste già
    if(!EdgeIsDupe(triMesh, extrema, original_id)){
        triMesh.Cell1DsId.push_back(eCount);
        triMesh.Cell1DsExtrema(0, eCount) = grid[i][j];
        triMesh.Cell1DsExtrema(1, eCount) = grid[i + 1][j + 1];
        eCount++;
    }

    // Crea le nuove facce dopo la triangolazione
    for (size_t i = 0; i < grid.size() - 1; i++){
        for (size_t j = 0; j <= i; j++){

            // Facce con la punta verso l'alto
            vector<unsigned int> v1 = {grid[i][j],grid[i + 1][j],grid[i + 1][j + 1]};
            vector<unsigned int> e1;
            // Memorizza i lati corrispondenti
            for(unsigned int v = 0; v < 3; v++){
                unsigned int from = v1[v];
                unsigned int to = v1[(v + 1) % 3];
                for(unsigned int k = 0; k < triMesh.Cell1DsId.size(); k++){
                    if((from == triMesh.Cell1DsExtrema(0, k) && to == triMesh.Cell1DsExtrema(1, k)) ||
                       (from == triMesh.Cell1DsExtrema(1, k) && to == triMesh.Cell1DsExtrema(0, k))){
                        e1.push_back(k);
                        break;
                    }
                }
            }
            // Aggiorna le variabili relative a Cell2Ds
            triMesh.Cell2DsId.push_back(fCount);
            triMesh.Cell2DsVertices.push_back(v1);
            triMesh.Cell2DsEdges.push_back(e1);
            fCount++;

            // Facce con la punta verso il basso
            if(i > 0 && j <= i - 1){
                vector<unsigned int> v2 = {grid[i][j],grid[i][j + 1],grid[i + 1][j + 1]};
                vector<unsigned int> e2;
                // Memorizza i lati corrispondenti
                for(unsigned int v = 0; v < 3; v++){
                    unsigned int from = v2[v];
                    unsigned int to = v2[(v + 1) % 3];
                    for(unsigned int k = 0; k < triMesh.Cell1DsId.size(); k++){
                        if((from == triMesh.Cell1DsExtrema(0, k) && to == triMesh.Cell1DsExtrema(1, k)) ||
                           (from == triMesh.Cell1DsExtrema(1, k) && to == triMesh.Cell1DsExtrema(0, k))){
                            e2.push_back(k);
                            break;
                        }
                    }
                }
                // Aggiorna le variabili relative a Cell2Ds
                triMesh.Cell2DsId.push_back(fCount);
                triMesh.Cell2DsVertices.push_back(v2);
                triMesh.Cell2DsEdges.push_back(e2);
                fCount++;
            }
        }
    }

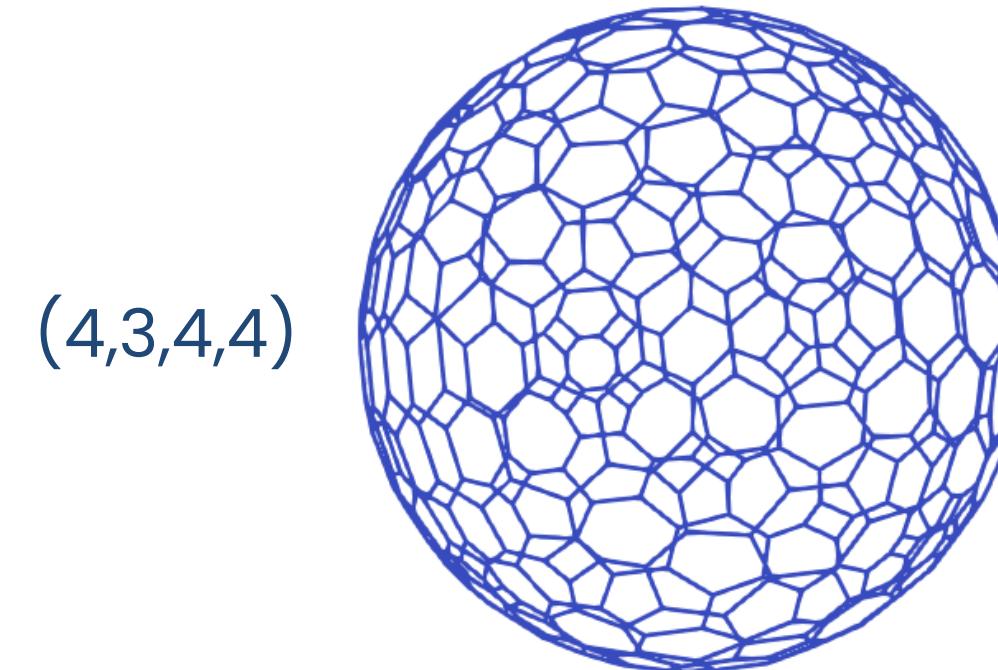
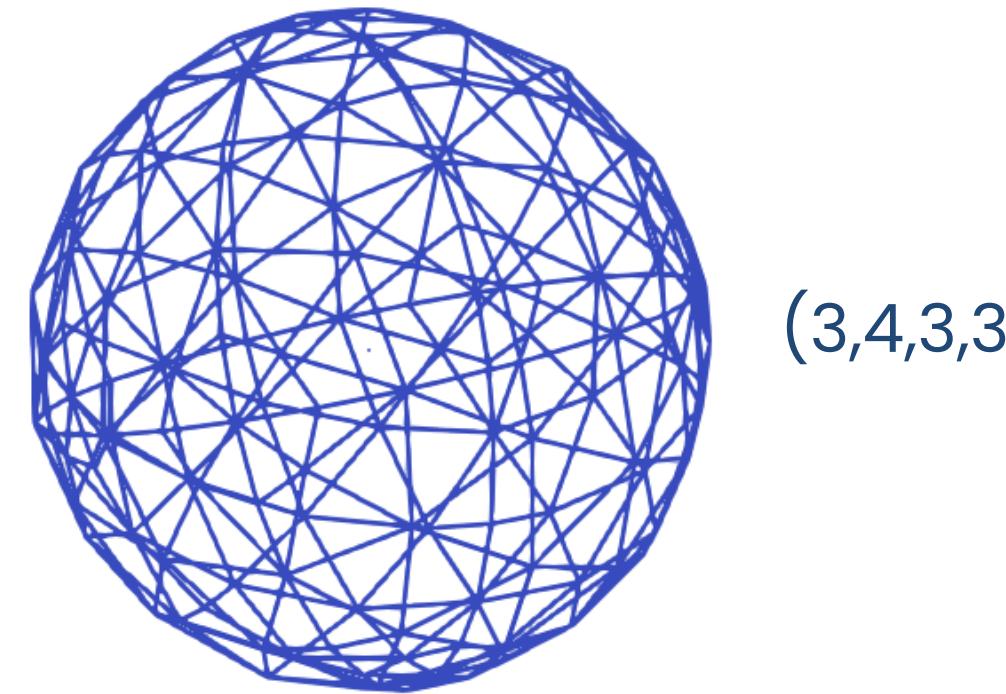
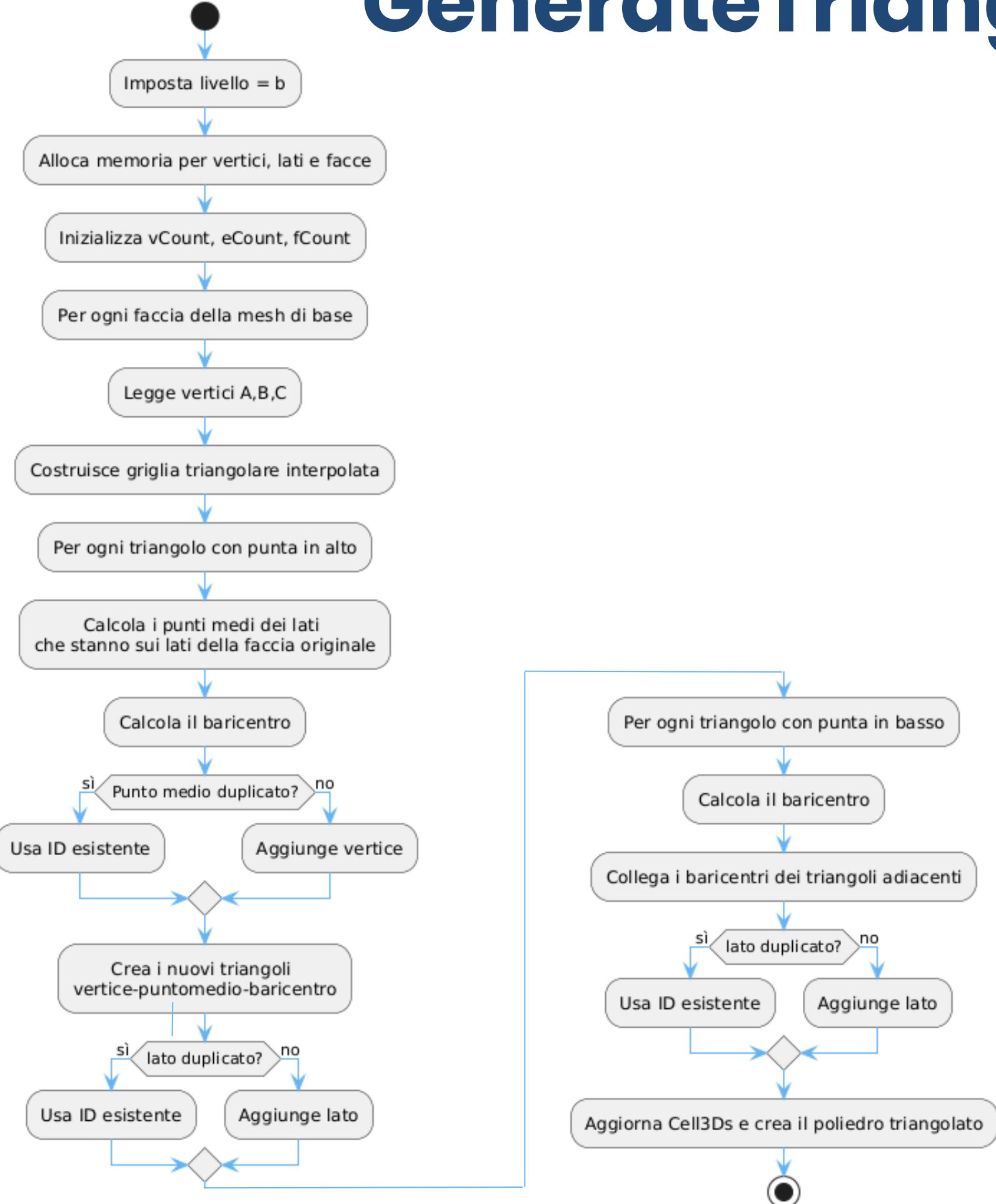
    // Crea il poliedro triangolato
    triMesh.Cell3DsId = {0};
    triMesh.Cell3DsVertices = triMesh.Cell0DsId;
    triMesh.Cell3DsEdges = triMesh.Cell1DsId;
    triMesh.Cell3DsFaces = triMesh.Cell2DsId;

    // Aggiorna i conteggi delle celle
    triMesh.NumCell0Ds = triMesh.Cell0DsId.size();
    triMesh.NumCell1Ds = triMesh.Cell1DsId.size();
    triMesh.NumCell2Ds = triMesh.Cell2DsId.size();
    triMesh.NumCell3Ds = 1;

    return true;
}

```

GenerateTriangulatedMesh2



```

// Funzione di triangolazione - POLIEDRI GEODETICI DI CLASSE II
bool GenerateTriangulatedMesh2(PolyhedralMesh& baseMesh, PolyhedralMesh& triMesh,
    const unsigned int& b, const unsigned int& c, // Parametri della suddivisione
    const Vector3i& triDimensions) // Dimensione di (V,E,F) della mesh triangolata
{
    unsigned int level = b; // Numero di suddivisioni laterali per triangolo (per la griglia base)

    // Inizializzazione della struttura dati della mesh triangolata
    triMesh.Cell0DsCoordinates.resize(3, triDimensions[0]);
    triMesh.Cell0DsId.reserve(triDimensions[0]);

    triMesh.Cell1DsId.reserve(triDimensions[1]);
    triMesh.Cell1DsExtrema.reserve(2, triDimensions[1]);

    triMesh.Cell2DsId.reserve(triDimensions[2]);
    triMesh.Cell2DsEdges.reserve(triDimensions[2]);
    triMesh.Cell2DsVertices.reserve(triDimensions[2]);

    // Contatori per vertici, spigoli e facce
    unsigned int vCount = 0;
    unsigned int eCount = 0;
    unsigned int fCount = 0;

    // Triangolazione per ogni faccia del poliedro iniziale
    for (unsigned int faceIdx = 0; faceIdx < baseMesh.Cell2DsId.size(); faceIdx++)
    {
        const auto& faceVerts = baseMesh.Cell2DsVertices[faceIdx];
        Vector3d A = baseMesh.Cell0DsCoordinates.col(faceVerts[0]);
        Vector3d B = baseMesh.Cell0DsCoordinates.col(faceVerts[1]);
        Vector3d C = baseMesh.Cell0DsCoordinates.col(faceVerts[2]);

        // Griglia di vertici base (come in GenerateTriangulatedMesh1)
        vector<vector<unsigned int>> grid_base_verts;
        for (unsigned int i = 0; i <= level; i++){
            vector<unsigned int> row;
            unsigned int original_id;
            Vector3d from = ((double)i / level) * B + ((double)(level - i) / level) * A;
            Vector3d to = ((double)i / level) * C + ((double)(level - i) / level) * A;
            for (unsigned int j = 0; j <= i; j++){
                Vector3d pos;
                if (i == 0){
                    pos = A;
                }
                else{
                    pos = ((double)j / i) * to + ((double)(i - j) / i) * from;
                }
                //pos = pos / pos.norm();
                if(!VertexIsDupe(triMesh, pos, original_id)){
                    triMesh.Cell0DsId.push_back(vCount);
                    triMesh.Cell0DsCoordinates.col(vCount) = pos;
                    row.push_back(vCount);
                    vCount++;
                }
                else{
                    row.push_back(original_id);
                }
            }
            grid_base_verts.push_back(row);
        }

        vector<vector<unsigned int>> barycenters_grid; // Griglia dei baricentri
        vector<vector<unsigned int>> barycenters_grid2; // Griglia dei baricentri 2
        vector<vector<vector<unsigned int>>> verts_barycenters_grid2; // Griglia coi vertici dei triangoli con la punta verso il basso

        // Ora per ogni triangolo creato con la triangolazione 1 applico la triangolazione 2
        for (unsigned int i = 0; i < level; i++) {
            vector<unsigned int> barycenters; // Vettore per segnare i baricentri dei triangoli verso l'alto della riga corrente
            vector<unsigned int> barycenters2; // Vettore per segnare i baricentri dei triangoli verso il basso della riga corrente
            vector<vector<unsigned int>> verts_barycenters_row2;

            for (unsigned int j = 0; j <= i; j++) {

                // Triangoli con la punto verso l'alto
                vector<unsigned int> triangleVertices1 = {grid_base_verts[i][j], grid_base_verts[i + 1][j], grid_base_verts[i + 1][j + 1]};

                // Prende le coordinate della faccia corrente ottenuta dalla triangolazione 1
                Vector3d p1_coord = triMesh.Cell0DsCoordinates.col(triangleVertices1[0]);
                Vector3d p2_coord = triMesh.Cell0DsCoordinates.col(triangleVertices1[1]);
                Vector3d p3_coord = triMesh.Cell0DsCoordinates.col(triangleVertices1[2]);

```

```

// Crea i nuovi triangoli vertice-puntomedio-baricentro (se sono previsti per questo triangolo)
vector<vector<unsigned int>> new_sub_triangles_1;

if(exists12){
    new_sub_triangles_1.push_back(std::vector<unsigned int>{triangleVertices1[0], mid12_id, barycenter_id});
    new_sub_triangles_1.push_back(std::vector<unsigned int>{mid12_id, triangleVertices1[1], barycenter_id});
}
if(exists23){
    new_sub_triangles_1.push_back(std::vector<unsigned int>{triangleVertices1[1], mid23_id, barycenter_id});
    new_sub_triangles_1.push_back(std::vector<unsigned int>{mid23_id, triangleVertices1[2], barycenter_id});
}
if(exists31){
    new_sub_triangles_1.push_back(std::vector<unsigned int>{triangleVertices1[2], mid31_id, barycenter_id});
    new_sub_triangles_1.push_back(std::vector<unsigned int>{mid31_id, triangleVertices1[0], barycenter_id});
}

unsigned int original_id2;

// Controlla se i lati esistono già prima di aggiungerli
for(const auto& new_verts : new_sub_triangles_1){
    vector<unsigned int> current_edges;
    for(unsigned int k = 0; k < 3; k++){
        Vector2i edge_extrema;
        edge_extrema[0] = new_verts[k];
        edge_extrema[1] = new_verts[(k + 1) % 3];

        if(EdgeIsDupe(triMesh, edge_extrema, original_id2)){
            current_edges.push_back(original_id2);
        }
        else {
            triMesh.Cell1DsId.push_back(eCount);
            triMesh.Cell1DsExtrema.col(eCount) = edge_extrema;
            current_edges.push_back(eCount);
            eCount++;
        }
    }

    // Aggiorna Cell2Ds
    triMesh.Cell2DsId.push_back(fCount);
    triMesh.Cell2DsVertices.push_back(new_verts);
    triMesh.Cell2DsEdges.push_back(current_edges);
    fCount++;
}

// Triangoli con la punta verso il basso (si parte dal secondo strato)
if (i > 0 && j < i) {
    vector<unsigned int> triangleVertices2 = {grid_base_verts[i][j], grid_base_verts[i][j + 1], grid_base_verts[i + 1][j + 1]};
    verts_barycenters_row2.push_back(triangleVertices2);

    // Prende le coordinate della faccia corrente ottenuta dalla triangolazione 1
    Vector3d p1_coord = triMesh.Cell1DsCoordinates.col(triangleVertices2[0]);
    Vector3d p2_coord = triMesh.Cell1DsCoordinates.col(triangleVertices2[1]);
    Vector3d p3_coord = triMesh.Cell1DsCoordinates.col(triangleVertices2[2]);

    // In questo caso salva solo le coordinate del barycentro
    Vector3d barycenter_pos = (p1_coord + p2_coord + p3_coord) / 3.0;
    barycenter_pos = barycenter_pos / barycenter_pos.norm();

    triMesh.Cell1DsId.push_back(vCount);
    triMesh.Cell1DsCoordinates.col(vCount) = barycenter_pos;
    unsigned int barycenter_id = vCount;
    barycenters2.push_back(vCount);
    vCount++;
}

// Aggiorna la griglia dei barycentri, necessaria per creare gli spigoli tra di essi
barycenters_grid.push_back(barycenters);
if(barycenters2.size() != 0) {
    barycenters_grid2.push_back(barycenters2);
    verts_barycenters_grid2.push_back(verts_barycenters_row2);
}
}

// Crea i collegamenti tra i barycentri (solo per i triangoli con la punta verso il basso)
for (unsigned int i = 0; i < barycenters_grid2.size(); i++) {
    for (unsigned int j = 0; j < barycenters_grid2[i].size(); j++) {

        // Prende gli id dei vertici della faccia a cui appartiene il barycentro
        unsigned int A = verts_barycenters_grid2[i][j][0];
        unsigned int B = verts_barycenters_grid2[i][j][1];
        unsigned int C = verts_barycenters_grid2[i][j][2];

        // Aggiunge i triangoli che si creano dal collegamento coi barycentri dei triangoli adiacenti
        vector<vector<unsigned int>> new_sub_triangles = {
            {A, barycenters_grid[i][j], barycenters_grid2[i][j]},
            {B, barycenters_grid[i][j], barycenters_grid2[i][j]},
            {A, barycenters_grid[i + 1][j], barycenters_grid2[i][j]},
            {C, barycenters_grid[i + 1][j], barycenters_grid2[i][j]},
            {B, barycenters_grid[i + 1][j + 1], barycenters_grid2[i][j]},
            {C, barycenters_grid[i + 1][j + 1], barycenters_grid2[i][j]}
        };

        unsigned int original_id;

        // Controlla se i lati esistono già prima di aggiungerli
        for(const auto& new_verts : new_sub_triangles){
            vector<unsigned int> current_edges;
            for(unsigned int k = 0; k < 3; k++){
                Vector2i edge_extrema;
                edge_extrema[0] = new_verts[k];
                edge_extrema[1] = new_verts[(k + 1) % 3];

                if(EdgeIsDupe(triMesh, edge_extrema, original_id)){
                    current_edges.push_back(original_id);
                }
                else {
                    triMesh.Cell1DsId.push_back(eCount);
                    triMesh.Cell1DsExtrema.col(eCount) = edge_extrema;
                    current_edges.push_back(eCount);
                    eCount++;
                }
            }

            // Aggiorna Cell2Ds
            triMesh.Cell2DsId.push_back(fCount);
            triMesh.Cell2DsVertices.push_back(new_verts);
            triMesh.Cell2DsEdges.push_back(current_edges);
            fCount++;
        }
    }
}

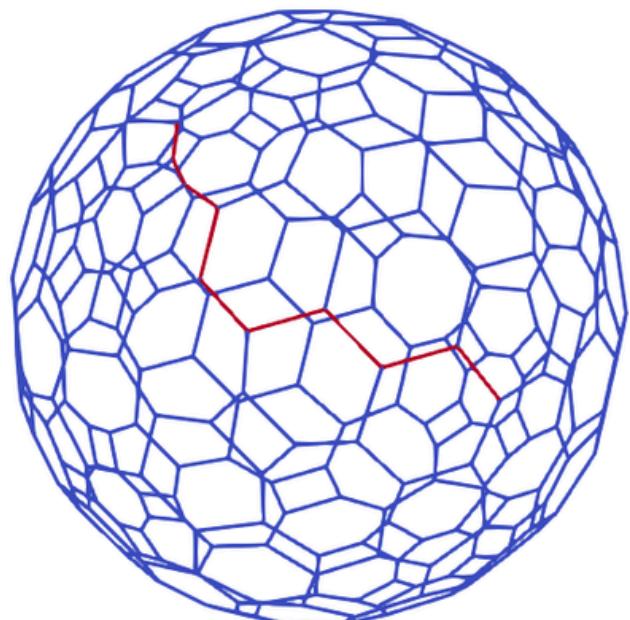
// Aggiorna Cell3DsId, Cell3DsVertices, Cell3DsEdges, Cell3DsFaces
triMesh.Cell3DsId = {0};
triMesh.Cell3DsVertices = triMesh.Cell1DsId;
triMesh.Cell3DsEdges = triMesh.Cell1DsId;
triMesh.Cell3DsFaces = triMesh.Cell2DsId;

triMesh.NumCell1Ds = triMesh.Cell1DsId.size();
triMesh.NumCell1Ds = triMesh.Cell1DsId.size();
triMesh.NumCell2Ds = triMesh.Cell2DsId.size();
triMesh.NumCell3Ds = 1;

return true;
}

```

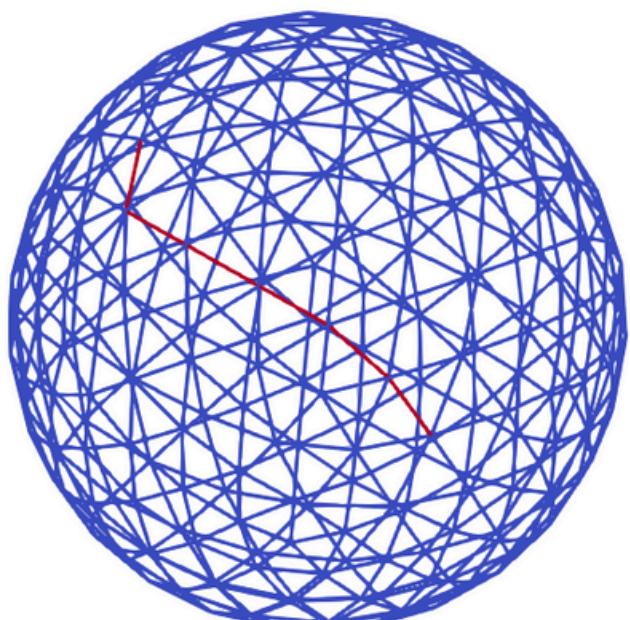
ShortestPath



(4,3,3,3,34,67)

Numero di archi nel cammino: 9

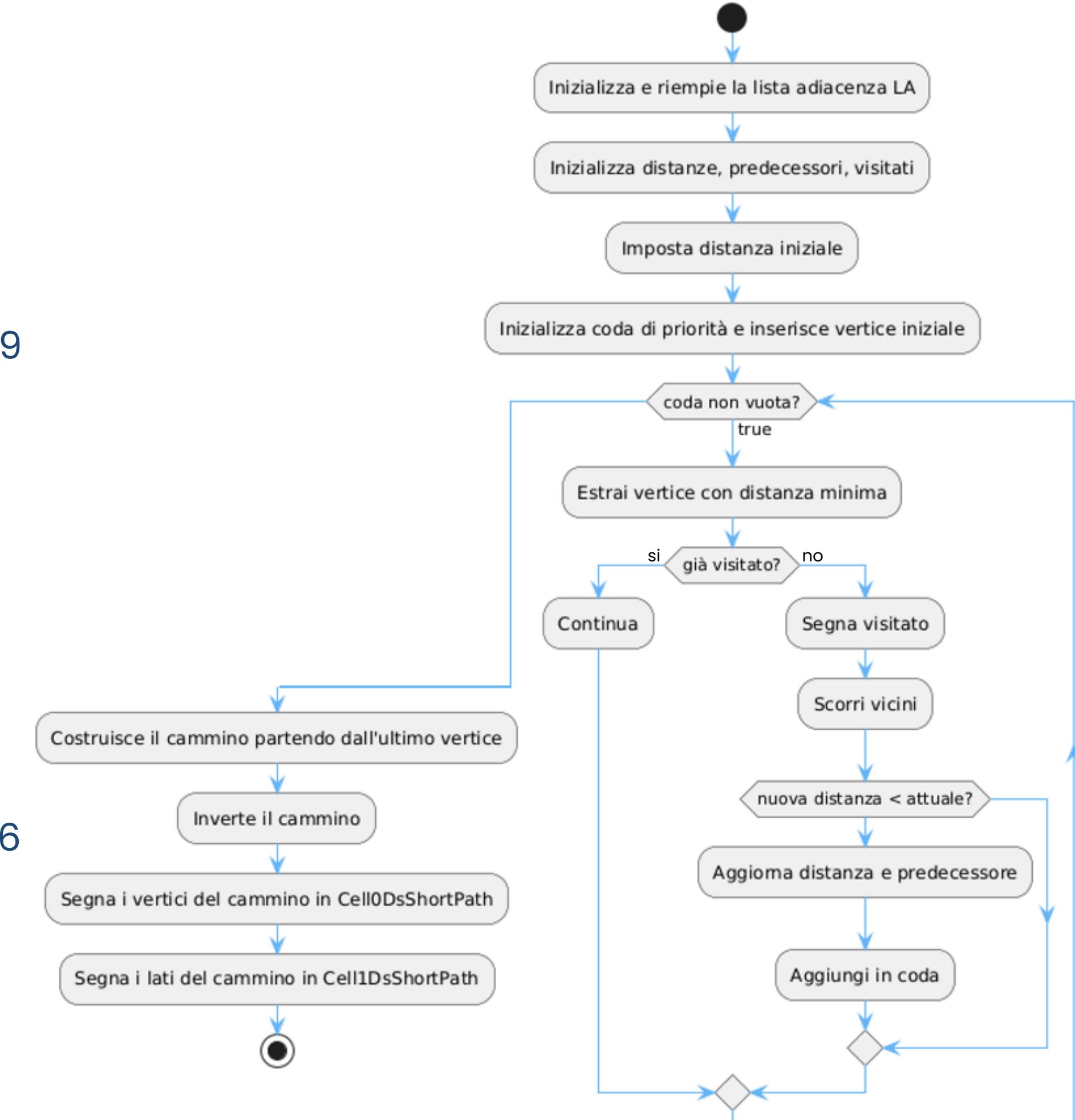
Lunghezza totale: 1.73502



(3,5,0,5,5,150)

Numero di archi nel cammino: 6

Lunghezza totale: 1.44074



```

// Funzione che calcola il cammino minimo tra due vertici in una mesh triangolata utilizzando Dijkstra
bool ShortestPath(PolyhedralMesh& mesh, unsigned int id_vertice_1, unsigned int id_vertice_2) {
    // controlla se gli ID dei vertici sono validi
    if (id_vertice_1 >= mesh.NumCell0Ds || id_vertice_2 >= mesh.NumCell0Ds) {
        cerr << "ID dei vertici non valido." << endl;
        return false;
    }

    // Inizializza il numero totale dei vertici della mesh
    unsigned int N = mesh.NumCell0Ds;

    // Inizializza la lista di adiacenza come vettore di vettori di coppie (vertice adiacente, peso).
    // L'i-esimo vettore corrisponde all'i-esimo vertice e contiene i vertici adiacenti e la relativa distanza
    vector<vector<pair<unsigned int, double>>> LA(N);

    // Viene riempita la lista di adiacenza considerando solo i lati della mesh
    for (unsigned int i = 0; i < mesh.NumCell1Ds; i++) {
        unsigned int v1 = mesh.Cell1DsExtrema(0, i); // Estremo 1 del lato
        unsigned int v2 = mesh.Cell1DsExtrema(1, i); // Estremo 2 del lato

        // Coordinate dei due estremi
        Eigen::Vector3d c1 = mesh.Cell0DsCoordinates.col(v1);
        Eigen::Vector3d c2 = mesh.Cell0DsCoordinates.col(v2);

        double weight = (c1 - c2).norm(); // Distanza tra i due estremi normalizzata = peso dell'arco

        // vengono aggiunti entrambi i sensi perché il grafo NON è orientato
        LA[v1].push_back({v2, weight});
        LA[v2].push_back({v1, weight});
    }

    // Inizializzazione Dijkstra
    vector<double> distance(N, numeric_limits<double>::infinity()); // Vettore delle distanze inizializzate ad infinito
    vector<int> pred(N, -1); // Vettore dei predecessori, inizializzati a -1
    vector<bool> visited(N, false); // Vettore dello stato dei vertici, inizializzati a non visitati

    distance[id_vertice_1] = 0.0; // La distanza del vertice iniziale da sè stesso è nulla

    // Inizializzazione coda di priorità
    priority_queue<pair<double, unsigned int>, vector<pair<double, unsigned int>>, greater<pair<double, unsigned int>>> pq;
    pq.push({0.0, id_vertice_1}); // Inserisce il vertice iniziale

    // Algoritmo di Dijkstra per il cammino minimo pesato
    while (!pq.empty()) {
        // Estraggo vertice con distanza minima
        unsigned int u = pq.top().second;
        pq.pop();

        if (visited[u]) continue; // Se già visitato, viene saltato
        visited[u] = true; // Viene marcato come visitato

        // Ciclo sui vertici adiacenti a u
        for (auto& [v, weight] : LA[u]) {
            if (visited[v]) continue; // I già visitati vengono saltati

            // se viene trovato un cammino più corto si aggiorna distanza e predecessore
            if (distance[u] + weight < distance[v]) {
                distance[v] = distance[u] + weight;
                pred[v] = u;
                pq.push({distance[v], v}); // v viene inserito nella coda in modo tale da essere visitato dopo
            }
        }
    }
}

// Se il nodo finale non ha predecessore, il cammino minimo non esiste
if (pred[id_vertice_2] == -1) {
    cerr << "Nessun cammino trovato da " << id_vertice_1 << " a " << id_vertice_2 << endl;
    return false;
}

// Ricostruzione del cammino minimo da vertice finale a vertice iniziale
// Viene utilizzata la condizione sul predecessore(pari a -1)
// in modo tale che il ciclo possa terminare una volta arrivato al vertice iniziale
vector<unsigned int> path;
int current_vertex = id_vertice_2;

while (current_vertex != -1) {
    path.push_back(current_vertex);
    current_vertex = pred[current_vertex];
}

// Inverte l'ordine in modo da avere il cammino effettivo
reverse(path.begin(), path.end());

// Inizializzazione delle proprietà della mesh
mesh.Cell0DsShortPath.resize(mesh.NumCell0Ds, 0);
mesh.Cell1DsShortPath.resize(mesh.NumCell1Ds, 0);

// Vengono marcati i vertici che appartengono al cammino
for (int v : path) {
    mesh.Cell0DsShortPath[v] = 1;
}

double length_tot = 0.0; // Somma totale delle distanze
unsigned int length_path = 0; // Numero totale di archi che compongono il cammino

// Vengono marcati i lati del cammino e ne viene calcolata la lunghezza
for (unsigned int i = 0; i < path.size() - 1; i++) {
    unsigned int u = path[i];
    unsigned int w = path[i + 1];

    for (unsigned int j = 0; j < mesh.NumCell1Ds; j++) {
        unsigned int v1 = mesh.Cell1DsExtrema(0, j);
        unsigned int v2 = mesh.Cell1DsExtrema(1, j);
        if ((v1 == u && v2 == w) || (v1 == w && v2 == u)) { // Il grafo non è orientato
            mesh.Cell1DsShortPath[j] = 1;
            Eigen::Vector3d c1 = mesh.Cell0DsCoordinates.col(v1);
            Eigen::Vector3d c2 = mesh.Cell0DsCoordinates.col(v2);
            length_tot += (c1 - c2).norm();
            length_path++;
            break;
        }
    }
}

cout << "Numero di archi nel cammino: " << length_path << endl;
cout << "Lunghezza totale: " << length_tot << endl;

return true;
}

```

The background features two large, semi-transparent blue shapes: a trapezoid on the left and a rounded rectangle on the right, both tilted diagonally.

Grazie per l'attenzione!