# Topological gift wrapping 2D

Link repository

Matteo Maraziti

Federico Tocci

Giacomo Scordino

# 1 STUDIO PRELIMINARE

Il topological gift wrapping è un algoritmo che produce un insieme di complessi di catene in 2D.
Data una qualsiasi collezione di poliedri cellulari la computazione può essere riassunta con i seguenti passaggi:

1. Estrarre i due scheletri dei poliedri;
2. Fondere in modo efficiente tutte le loro 2-celle;
3. Calcolare su ogni 2-cella il suo complesso di catene locale.

Con tali premesse, l'obiettivo del presente elaborato è stato quello di effettuare una analisi preliminare del codice a disposizione, individuando i compiti principali che l'algoritmo svolge, le dipendenze fra le varie funzione che lo compongono e determinare eventuali criticità su cui è necessario intervenire.

## 1.1 IL LINGUAGGIO JULIA

L'algoritmo appena introdotto utilizza Julia come linguaggio di programmazione. Esso è stato creato con l'intento di garantire alte prestazioni, sfruttando a pieno le potenzialità del calcolo parallelo. È possibile utilizzare primitive che permettono di sfruttare a pieno i *core* delle macchine sulle quali viene messo in esecuzione il codice Julia, grazie al meccanismo di multi-threading.
Julia può inoltre generare codice nativo per GPU, risorsa che permette di abbattere ulteriormente i tempi di esecuzione dell'algoritmo.

## 1.2 FUNZIONAMENTO

L'algoritmo è utilizzato localmente su 2-cella per essere decomposta, e invece utilizzato globalmente per generare le 3-celle della partizione dello spazio.

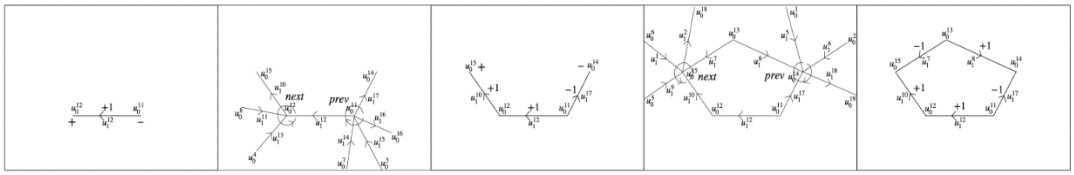

Figura 1 : Estrazione di 1 ciclo minimale

Per ogni elemento (1-scheletro) calcolo *il bordo* ottenendo i due vertici, per ciascun vertice calcolo il cobordo, ovvero individuo gli altri elementi (1-scheletro) con un vertice coincidente (questo passaggio viene effettuato tramite valori matriciali). A questo punto si isolano due elementi tra quelli individuati formando così una catena e si ripete l'algoritmo sugli elementi della catena appena calcolata. L'obiettivo di ciascuna iterazione è quello di individuare una porzione nel piano (ovvero la 1-catena di bordo)Figura 1.

### 1.2.1 ILLUSTRAZIONE DELLO PSEUDOCODICE

Lo pseudocodice in Figura 2 è il riassunto dell'algoritmo TGW in uno spazio generico di D-dimensionale.

L'algoritmo prende in input una matrice sparsa di dimensioni "m×n" e restituisce una matrice dal dominio delle D-catene a quello dei (d-1) cicli orientati.

1. Initialization: $m, n = [\partial_{d-1}].shape$ ; $marks = zeros(n)$; $[\partial_d^+] = []$ ;
2. **while** sum($marks$) $< 2n$ **do**
   (a) select the $(d-1)$-cell seed of the column extraction
   (b) compute boundary $c_{d-2}$ of seed cell
   (c) **until** boundary $c_{d-2}$ becomes empty **do**
      i. $corolla = []$
      ii. **for each** "stem" cell $\tau \in c_{d-2}$ **do**
         a. compute the $\tau$ coboundary
         b. compute the new *petal* cell
         c. orient *petal* and insert it in $corolla$
      iii. insert $corolla$ cells in current $c_{d-1}$
      iv. compute again the $c_{d-2}$ boundary of $c_{d-1}$
   (d) increment the $marks$ counters of used cells
   (e) append a new column to $[\partial_d^+]$

Figura 2: pseudocodice

## 1.3 FUNZIONI INTERNE PRINCIPALI

### 1.3.1 PLANAR ARRANGEMENT

```
function planar_arrangement_1( V, copEV,
                sigma::Lar.Chain=spzeros(Int8, 0),
                return_edge_map::Bool=false,
                multiproc::Bool=false)
        # data structures initialization
        edgenum = size(copEV, 1)
        edge_map = Array{Array{Int, 1}, 1}(undef,edgenum)
        rV = Lar.Points(zeros(0, 2))
        rEV = SparseArrays.spzeros(Int8, 0, 0)
        finalcells_num = 0

        # spaceindex computation
        model = (convert(Lar.Points,V'),Lar.cop2lar(copEV))
        bigPI = Lar.spaceindex(model::Lar.LAR)

    # multiprocessing of edge fragmentation
    if (multiproc == true)
        in_chan = Distributed.RemoteChannel(()->Channel{Int64}(0))
        out_chan = Distributed.RemoteChannel(()->Channel{Tuple}(0))
        ordered_dict = SortedDict{Int64,Tuple}()
        @async begin
            for i in 1:edgenum
                put!(in_chan,i)
            end
            for p in distributed.workers()
                put!(in_chan,-1)
            end
        end
        for p in distributed.workers()
            @async Base.remote_do(frag_edge_channel, p, in_chan, out_chan, V, copEV, bigPI)
        end
        for i in 1:edgenum
            frag_done_job = take!(out_chan)
            ordered_dict[frag_done_job[1]] = frag_done_job[2]
        end
        for (dkey, dval) in ordered_dict
            i = dkey
            v, ev = dval
            newedges_nums = map(x->x+finalcells_num, collect(1:size(ev, 1)))
            edge_map[i] = newedges_nums
            finalcells_num += size(ev, 1)
            rV, rEV = Lar.skel_merge(rV, rEV, v, ev)
        end
    else
        # sequential (iterative) processing of edge fragmentation
        for i in 1:edgenum
            v, ev = frag_edge(V, copEV, i, bigPI)
            newedges_nums = map(x->x+finalcells_num, collect(1:size(ev, 1)))
            edge_map[i] = newedges_nums
            finalcells_num += size(ev, 1)
            rV = convert(Lar.Points, rV)
            rV, rEV = Lar.skel_merge(rV, rEV, v, ev)
        end
    end
    # merging of close vertices and edges (2D congruence)
    V, copEV = rV, rEV
    V, copEV = merge_vertices!(V, copEV, edge_map)
        return V,copEV,sigma,edge_map
end
```

L’obiettivo è partizionare un complesso cellulare passato come parametro. Un complesso cellulare è partizionato quando l'intersezione di ogni possibile coppia di celle del complesso è vuota e l'unione di tutte le celle è l'intero spazio euclideo.

### 1.3.2 MERGE_VERTICES

```
function merge_vertices!(V::Lar.Points, EV::Lar.ChainOp, edge_map, err=1e-4)
    vertsnum = size(V, 1)
    edgenum = size(EV, 1)
    newverts = zeros(Int, vertsnum)
    # KDTree constructor needs an explicit array of Float64
    V = Array{Float64,2}(V)
    kdtree = KDTree(permutedims(V))

    # merge congruent vertices
    todelete = []
    i = 1
    for vi in 1:vertsnum
        if !(vi in todelete)
            nearvs = Lar.inrange(kdtree, V[vi, :], err)
            newverts[nearvs] .= i
            nearvs = setdiff(nearvs, vi)
            todelete = union(todelete, nearvs)
            i = i + 1
        end
    end
    nV = V[setdiff(collect(1:vertsnum), todelete), :]

    # merge congruent edges
    edges = Array{Tuple{Int, Int}, 1}(undef, edgenum)
    oedges = Array{Tuple{Int, Int}, 1}(undef, edgenum)
    for ei in 1:edgenum
        v1, v2 = EV[ei, :].nzind
        edges[ei] = Tuple{Int, Int}(sort([newverts[v1], newverts[v2]]))
        oedges[ei] = Tuple{Int, Int}(sort([v1, v2]))
    end
    nedges = union(edges)
    nedges = filter(t->t[1]!=t[2], nedges)
    nedgenum = length(nedges)
    nEV = spzeros(Int8, nedgenum, size(nV, 1))
    # maps pairs of vertex indices to edge index
    etuple2idx = Dict{Tuple{Int, Int}, Int}()
    # builds `edge_map`
    for ei in 1:nedgenum
        nEV[ei, collect(nedges[ei])] .= 1
        etuple2idx[nedges[ei]] = ei
    end
    for i in 1:length(edge_map)
        row = edge_map[i]
        row = map(x->edges[x], row)
        row = filter(t->t[1]!=t[2], row)
        row = map(x->etuple2idx[x], row)
        edge_map[i] = row
    end
    # return new vertices and new edges
    return Lar.Points(nV), nEV
end
```

Si occupa di fondere vertici congruenti e bordi congruenti, assegnare a coppie di indici di vertici indici di bordo e costruire una mappa dei bordi.

### 1.3.3 FRAG_EDGE

```
function frag_edge(V, EV::Lar.ChainOp, edge_idx::Int, bigPI)
    alphas = Dict{Float64, Int}()
    edge = EV[edge_idx, :]
    verts = V[edge.nzind, :]
    for i in bigPI[edge_idx]
        if i != edge_idx
            intersection = Lar.Arrangement.intersect_edges(
                V, edge, EV[i, :])
            for (point, alpha) in intersection
                verts = [verts; point]
                alphas[alpha] = size(verts, 1)
            end
        end
    end
    alphas[0.0], alphas[1.0] = [1, 2]
    alphas_keys = sort(collect(keys(alphas)))
    edge_num = length(alphas_keys)-1
    verts_num = size(verts, 1)
    ev = SparseArrays.spzeros(Int8, edge_num, verts_num)
    for i in 1:edge_num
        ev[i, alphas[alphas_keys[i]]] = 1
        ev[i, alphas[alphas_keys[i+1]]] = 1
    end
    return verts, ev
end
```

Si occupa della frammentazione dei bordi in EV usando l'indice spaziale bigPI.

## 1.4 CONCLUSIONI PER FUTURA OTTIMIZZAZIONE

Analizzando il codice nel dettaglio, è possibile evidenziare che in alcuni passi dell'algoritmo è stato implementato il calcolo parallelo e distribuito. Infatti, nella funzione "*planar_arrangement_1*", la frammentazione dei bordi può essere effettuata tramite il calcolo asincrono.

Continuando l'analisi del codice ed osservando accuratamente le dipendenze presenti risulta opportuno implementare modifiche con l'obiettivo di migliorare scalabilità, modificabilità e prestazioni di porzioni dello stesso, riducendo l'accoppiamento tra i moduli presenti fattorizzando il codice e continuando ad implementare forme di calcolo parallelo e distribuito. In particolare, alcune di queste modifiche dovranno coinvolgere il codice relativo alla funzione "*merge_vertices!*", presentata in precedenza. Infatti, ad essa sono assegnate numerose *task* che possono essere suddivise in diverse sotto funzioni.