Relazione PCS

Elena Bongiovanni, Chiara Grosso, Elena Perotti

1 Introduzione

In questo progetto abbiamo importato delle fratture da alcuni file e ne abbiamo determinato le cosiddette tracce, ossia le intersezioni tra fratture distinte. In un secondo momento abbiamo proceduto a tagliare le figure, prima utilizzando le tracce passanti ed in seguito quelle non passanti.
In generale abbiamo cercato di risolvere il minor numero possibile di sistemi lineari, poiché si tratta di operazioni molto costose a livello computazionale e di utilizzare i vettori in tutti i casi in cui non fosse necessario fare inserimenti nelle posizioni centrali, poiché sono strutture a cui si puòaccedere con un costo costante $O(1)$.

2 Strutture dati e relativi metodi

Abbiamo creato delle struct per inserirvi i dati relativi alle principali entità geometriche presenti nel programma.

2.1 Trace

Per ogni traccia abbiamo memorizzato in questa struct l'identificatore, gli identificatori delle fratture che la generano, le coordinate, la lunghezza, la retta a cui la traccia appartiene ed il punto di applicazione di tale retta. In tutti i casi in cui fosse necessario memorizzare più informazioni abbiamo scelto di utilizzare dei vettori anziché le liste poiché si tratta di dati di dimensione fissata a cui è possibile accedere in un tempo costante.

2.2 Fracture

Per ogni frattura sono stati memorizzati l'identificatore, il numero e le coordinate dei vertici, il baricentro, due liste per le tracce passanti e non passanti, una mappa *tips* che associa all'identificatore delle tracce un booleano che permette di capire se la traccia associata sia passante o meno, il numero di fratture, gli identificatori di lati e vertici, i coefficienti del piano a cui la frattura appartiene, una mappa che associa agli identificatori delle tracce che tagliano la frattura un booleano che indica se il taglio sia su un lato o meno. In questo caso abbiamo utilizzato dei vettori per implementare i dati la cui dimensione era nota a priori,

delle liste per quelli con una dimensione che è stata determinata soltanto dopo l'esecuzione del programma e delle mappe per quelli per cui risultasse funzionale avere tutte le informazioni associate ad un identificatore.

2.2.1 maxDist

Questo metodo permette di calcolare la distanza massima dei vertici di una data frattura dal baricentro della stessa e consta essenzialmente di due parti:

- un ciclo sui vertici della figura,
- la determinazione, ad ogni iterazione, della distanza del vertice i-esimo dal baricentro (per cui sono necessari tre sottrazioni ed elevamenti a potenza, tre somme ed un'estrazione di radice).

Inoltre, se il valore della distanza determinato al passo corrente risulta quello massimo trovato fino a quel momento, il valore r viene aggiornato. Il costo totale risulta dunque essere $O(V)$ dove V è il numero di vertici.

2.2.2 calcoloPiano

Il metodo determina i coefficienti direttori del piano a cui la frattura passatagli in input appartiene ed ha un costo costante dovuto alle seguenti operazioni:

- inizializzazione di due vettori 3D, con costo $O(1)$,
- determinazione di due lati attraverso 6 sottrazioni,
- calcolo del prodotto vettoriale mediante 6 sottrazioni e 6 moltiplicazioni,

Quindi, per ogni frattura, sono richieste circa 20 operazioni.
Inizialmente avevamo normalizzato i lati trovati e calcolato il prodotto vettoriale tra i versori così ottenuti per avere gli stessi coefficienti nei piani da confrontare, ma abbiamo eliminato questo passaggio poiché per verificare il parallelismo tra due piani è sufficiente che il prodotto tra i vettori ad essi ortogonali sia nullo.

2.3 FractureMesh

Questa struct rappresenta una mesh di fratture e fornisce strumenti per gestire un insieme di fratture e le loro interazioni. Vengono memorizzati il numero totale di fratture, un vettore contenenti gli identificatori di tutte le fratture, un vettore contenente tutte le fratture della mesh ed una mappa che associa gli identificatori delle tracce alle corrispondenti strutture *Trace*.

2.3.1 printingtraces

Questo metodo crea il nome del file di output basandosi su quello del file in input e scrive il numero di tracce nonché le informazioni relative a ciascuna traccia sul file designato. Sono presenti due cicli annidati che vengono eseguiti 6 volte complessivamente per ogni iterazione di quello esterno e, detto N il numero di tracce in *MapTrace*, poiché il ciclo esterno gira invece N volte, il costo complessivo è $O(N)$.

2.3.2 printingfractures

Il metodo è progettato per stampare le informazioni sulle fratture contenute in *MapFractures* su un file, elencando nel dettaglio le tracce passanti e non passanti per ciascuna frattura. La determinazione del nome del file di output ha un costo $O(m)$, dove m è la lunghezza della stringa *file*, le operazioni di apertura e chiusura dei file hanno costo costante $O(1)$ e risultano quindi trascurabili.
Infine è presente un ciclo sulle fratture della mappa *MapFractures*, all'interno del quale vi sono un ciclo sulle tracce passanti ed uno su quelle non passanti. Definiamo F il numero di fratture, P quello di tracce passanti e N quello di tracce non passanti. Supponendo che la lunghezza della stringa *file* sia trascurabile rispetto al numero di iterazioni del ciclo appena descritto, la complessità computazionale del metodo risulta $O(F * (P + N))$.

2.4 Cell0d

Le *Cell0d* rappresentano dei punti nella mesh e sono caratterizzate da un identificatore, da un vettore contenente le coordinate, da un booleano che indica se il punto è già stato elaborato e da un elenco degli identificatori delle *Cell2d* che toccano la cella in questione. Sono inoltre presenti un costruttore di default ed uno parametrizzato che permette di inizializzare *id* e *coordinates* coi valori forniti.

2.5 Cell1d

Le *Cell1d* rappresentano dei segmenti nella mesh e sono caratterizzate da un identificatore, da un vettore contenente le coordinate degli estremi, da una lista degli identificatori delle *Cell2d* a cui la cella in questione appartiene, da un booleano che indica se sono già state elaborate e da un vettore in cui vengono salvate le *Cell1d* in cui la cella verrà divisa dopo le operazioni di taglio. Sono inoltre presenti un costruttore di default ed uno parametrizzato che permette di inizializzare *id* e *extremes* e *old* coi valori forniti.

2.6 Cell2d

Le *Cell2d* rappresentano dei poligoni nella mesh e sono caratterizzate da un identificatore, da un vettore contenente le coordinate dei vertici ed uno conte-

nenti i lati, dal numero dei vertici, da un booleano che indica se sono già state elaborate e da un identificatore della frattura corrispondente.

2.6.1 convertFracture

Il metodo permette di convertire una *Fracture* in una *Cell2d* servendosi di una rappresentazione che sfrutta anche le *Cell1d* e le *Cell0d*. Abbiamo introdotto questo metodo per facilitare le operazione di manipolazione e analisi sulla mesh una volta operati i tagli. Nel corpo del metodo sono presenti due cicli che iterano V volte, dove V è il numero di vertici della *Cell2d* e alcune operazioni di assegnazione (aventi costo costante). Il costo totale è quindi $O(V)$.

2.7 PolygonalMesh

Questa struct è una mesh poligonale, ossia una collezione di celle poligonali (delle tipologie descritte ai paragrafi precedenti) organizzate in modo da poter descrivere strutture geometriche complesse. Oltre alle strutture relative alle *Cell0d, Cell1d* e *Cell2d* è stato memorizzato il numero di celle di ciascun tipo.

2.7.1 addFirstElement

Questo metodo aggiunge una prima *Cell2d* alla relativa mesh ed aggiorna tutti i dati relativi in *PolygonalMesh*. Le operazioni di aggiornamento sono effettuate mediante dei cicli, ciascuno dei quali viene eseguito V volte, dove V è il numero di vertici ed è ovviamente coincidente con quelllo di lati. Il costo complessivo, dato che le assegnazioni hanno un costo trascurabile rispetto a quello dei cicli, è dunque $O(V)$.

2.7.2 addingStuff

Il metodo ha lo scopo di aggiungere nuove celle alla mesh a partire da celle esistenti che sono state tagliate. Definiamo i la dimensione del vettore *idLatitagliati*, f quella di *forming*, d quella di *forming0d* e c il numero di *Cell2d* interessate da tagli. L'aggiornamento delle celle tagliate ha un costo $O(i)$, la creazione delle nuove *Cell2d* e delle relative relazioni topologiche hanno costo costante $O(1)$, l'aggiunta delle nuove *Cell0d* e *Cell1d* alle relative mappe ha un costo $O(f)$ e $O(d)$ rispettivamente. Infine aggiornare le *Cell2d* che vengono tagliate, ipotizzando che il numero medio di *Cell2d* toccate da una *Cell1d* sia costante e che il ciclo interno abbia quindi un costo $O(1)$ (indipendentemente dal valore assunto da c) per ogni iterazione di quello esterno, ha un costo $O(i)$. Pertanto il metodo ha una complessità computazionale risultante pari a $O(i + f + d)$.

3 Funzioni usate e relativo costo computazionale

3.1 ImportFR_data

Abbiamo creato ImportFR_data per leggere i dati dai file in input e salvarli nelle struct *FractureMesh* e *Fracture*. Inoltre nel corpo di questa funzione i baricentri di ciascuna frattura vengono calcolati e inseriti nella relativa struct. Fondamentalmente i passi in cui la funzione si articola sono:

- apertura del file e controlli iniziali, con un costo pari a $O(1)$,
- saltare le righe di commenti, il cui costo risulta $O(r)$, dove r è il numero delle suddette righe,
- lettura del numero di fratture N e resize del vettore *FractureId*, in cui l'operazione prevalente è la seconda, avente un costo $O(N)$,
- lettura e analisi di ciascuna frattura, con un costo di $O(N)$, per processare gli identificatori e il numero di vertici,
- lettura dei vertici, calcolo del baricentro e memorizzazione delle informazioni ottenute nella struct *Fracture*, operazioni risultanti in costo $O(numvertices)$,
- memorizzazione dei dati della singola frattura nella struct *FractureMesh*, avente un costo costante $O(1)$ per ciascuna frattura, per un totale di $O(N)$.

Il costo computazionale risulta quindi lineare in N.

3.2 findIntersections

Questa funzione determina le tracce presenti all'interno di ciascuna frattura suddividendole in passanti e non passanti ed ordinando ciascuna di queste partizioni per lunghezza decrescente.
Il corpo è composto da un ciclo esterno sugli identificatori delle fratture in cui viene determinata l'equazione del piano individuato da ciascuna frattura attraverso la funzione *"calcoloPiano"*. Vi è poi un ciclo interno sugli id successivi rispetto a quello in questione all'interno del quale vengono calcolati i piani individuati dalle fratture successive e vengono fatte delle verifiche che permettono di determinare se possa esistere un' intersezione tra i due piani.
Sono pertanto state escluse le casistiche di fratture complanari (fatto salvo il caso in cui si intersechino lungo un lato, che viene analizzato attraverso *"sameLine"*) o giacenti su piani paralleli. Sono inoltre state escluse le coppie di fratture i cui baricentri avessero una distanza maggiore della somma dei raggi delle circonferenze definite come

$$C_i := \{x \in R^3 \ t.c. \ d(x, barycentre_i) = \max_{v_j}\{d(v_j, barycentre_i)\},$$

dove v_j è un vertice di F_i.
Adottando questi accorgimenti è possibile evitare molti calcoli che aumenterebbero il tempo necessario per l'esecuzione del programma.

Nell'eventualità in cui nessuna delle condizioni esposte sopra si sia verificata si procede a determinare l'equazione della retta di intersezione tra i piani attraverso *"PALUSolver"* e successivamente si calcolano le eventuali intersezioni di tale retta con le fratture attraverso la funzione *"intersezionipoligonoretta"*.
Nel caso in cui vengano trovate delle intersezioni si procede a ordinarle in base al primo elemento e successivamente a trovare le coordinate della traccia con la funzione *"intersezioniSuRetta"*. I dati delle tracce così determinate vengono quindi inseriti nella struct *Trace*.
Si procede dunque definendo le nuove tracce formatesi con *"defNewTrace"* creando per ciascuna frattura le liste delle tracce passanti e non e ordinandole per lunghezza decrescente. Siano N il numero di fratture e V il numero medio di vertici per ogni frattura. La presenza del ciclo annidato e della funzione *sort*, che risultano le componenti che gravano maggiormente sul tempo di esecuzione della funzione, fa sì che il costo risultante sia $O(N^2(V^2 + Vlog(V))$.

3.3 sameLine

La funzione verifica se una porzione di retta coincide con il vettore congiungente due vertici successivi della frattura. Nello specifico viene calcolata la norma della retta fornita in input e la normalizzata viene memorizzata in *rettaN*. Vi è inoltre un ciclo su tutti i vertici della frattura e per ogni coppia di vertici consecutivi viene determinato e normalizzato il segmento che li unisce. Infine le componenti normalizzate della retta e della congiungente i vertici vengono confrontate e se la loro differenza è minore della tolleranza si assume che le due linee coincidano, il vettore "coordinate" viene aggiornato e il valore da restituire viene settato a *true*.
Sia V il numero di vertici della frattura. Poiché il calcolo della norma di un vettore di dimensione costante ha costo fisso, così come le operazioni aritmetiche e di confronto, la parte più costosa a livello computazionale è il ciclo *for*, che ha costo lineare in V. Complessivamente la funzione ha costo $O(V)$.

3.4 onSegment

La funzione verifica se un punto appartiene alla retta passante per i due punti a e b in input. Si utilizza un parametro t per fare un' interpolazione lineare del punto all'interno del segmento ab. Il costo è costante poiché il numero di operazioni eseguite è indipendente dal numero di dati in input.

3.5 intersezioniSuRetta

Questa funzione consente di trovare le intersezioni tra due segmenti *s1* e *s2* appartenenti alla stessa retta e calcolare le coordinate della traccia che si forma dalla loro intersezione. Inizialmente i segmenti vengono ordinati in modo tale che il primo elemento sia quello con ascissa minore, così da ridurre le operazioni da svolgere al passo successivo. In seguito, attraverso dei semplici confronti tra le coordinate degli estremi, si determina l'eventuale intersezione e si trovano

così le coordinate degli estremi della traccia. Poiché l'accesso agli elementi di un vettore di lunghezza fissata e le operazioni di confronto hanno costo costante, il costo totale è $O(1)$.

3.6 dist

Questa funzione calcola la distanza tra i punti in input ed è composta da un ciclo *for* che viene eseguito tre volte. All'interno del ciclo si determina il quadrato della differenza tra le coordinate di egual posizione dei due punti e si aggiorna il valore della somma aggiungendo quello appena calcolato. Infine, determinando la radice quadrata della somma trovata in precedenza, si ottiene il valore della distanza cercato. Il numero di operazioni richiesto è dunque costante ed il costo risulta $O(1)$.

3.7 PALUSolver

Abbiamo scelto di utilizzare una fattorizzazione PALU della matrice per risolvere il sistema lineare poiché, in assenza di ulteriori informazioni sulla natura della matrice stessa, questo metodo di soluzione risulta quello computazionalmente più efficiente avendo un costo di $n^3/3$ per matrici quadrate di dimensione n.

3.8 intersezionipoligonoretta

La funzione ha lo scopo di determinare le intersezioni tra una frattura ed una retta (definita da una direzione ed un punto di applicazione) e restituisce gli eventuali punti di intersezione tra le due entità o un booleano *onEdge* se una porzione di retta coincide con un lato della frattura.
Preliminarmente si verifica che nessun lato coincida con una parte della retta e in seguito, per ogni lato del poligono, vengono eseguite le operazioni seguenti:

- determinazione del vettore che rappresenta il lato,
- calcolo del prodotto vettoriale tra la direzione della retta e il vettore che descrive il lato,
- calcolo del parametro *alfa*, che rappresenta la posizione del punto di intersezione lungo il lato,
- se $alfa \in (0 - tol, 1 + tol)$ l'intersezione trovata è valida in quanto combinazione convessa dei vertici
- calcolo dell coordinate del punto di intersezione

Poiché una retta può intersecare un poligono in al più due punti, se le intersezioni trovate sono due il ciclo descritto in precedenza viene interrotto. Si verifica poi se le intersezioni determinate abbiano distanza maggiore della tolleranza, vale a dire che non risultino pressoché coincidenti, nel cui caso si tratterebbe di

un'intersezione in un vertice ed una delle intersezioni trovate andrebbe eliminata. Il passo più oneroso dal punto di vista computazionale è la chiamata della funzione *sameLine*, quindi se V è il numero di vertici della frattura processata, il costo risulta $O(V)$.

3.9 compareFirtsElement

La funzione confronta il primo elemento dei due vettori passati in input e restituisce "true" se la differenza tra i due risulta essere minore della tolleranza fissata. Poiché gli elementi di un vettore sono accessibili in un tempo costante e le sottrazioni hanno anch'esse costo costante, il costo risultantè è $O(1)$.

3.10 defNewTrace

La funzione definisce e gestisce una nuova traccia all'interno della struttura dati *FractureMesh*, che rappresenta l'insieme delle fratture. Inizialmente viene calcolata la distanza tra gli estremi della traccia, in seguito la lunghezza e l'identificatore della traccia vengono inseriti nella struct *Trace*, mentre nelle struct relative alle fratture passate in input viene incrementato il numero di tracce.
Valutando il valore assoluto della diffferenza tra la lunghezza della traccia in questione e le distanze *d1* e *d2* passate in input è possibile stabilire se la traccia in questione sia passante o meno per *f1* e *f2* rispettivamente.
Il passo col costo computazionale maggiore è l'inserimento della traccia t nella mappa di tracce *MapTrace*, che ha costo $O(logn)$ se n è il numero di elementi della mappa.

3.11 orderLen

La funzione ha lo scopo di confrontare elementi di tipo *Trace* e restituisce *true* se il primo elemento in input ha lunghezza maggiore o uguale a quella del secondo. Il costo è $O(1)$ perché si tratta di un semplice confronto.

3.12 newpolygon

La funzione prende in input una struttura dati *FractureMesh*, che contiene una serie di fratture. Il suo compito è quello di creare e restituire un vettore di *PolygonalMesh*, ciascuno dei quali costituisce una rappresentazione poligonale delle fratture presenti nella mesh. Dopo l'inizializzazione delle strutture necessarie a contenere i dati la funzione utilizza il metodo *convertFracture* per creare una prima *Cell2d* e la aggiunge alla *PolygonalMesh*. Si itera poi sulle tracce e si tagliano le fratture mediante *cuttingfractures*. Durante il processo vengono create nuove celle per rappresentare le modifiche e le nuove entità vengono aggiunte alla mesh. Dopo che tutte le fratture sono state processate la funzione restituisce un vettore *newfigures* contenente tutte le nuove rappresentazioni poligonali

delle fratture presenti nella mesh. Il costo totale della funzione *newpolygon* può essere approssimato a $O(n * (m_{pas} + m_{nonpas} + v_{tot} + k_{tot})$, dove:

- n: Numero di fratture in mesh.
- m_{pas} : Somma delle dimensioni delle liste *listPas* di tutte le fratture.
- m_{nonpas}: Somma delle dimensioni delle liste *listNonpas* di tutte le fratture.
- v_{tot} : Numero totale di vertici delle *Cell2d* attraversate dalle tracce.
- k_{tot} : Numero totale di nuove entità *Cell2d, Cell1d, Cell0d* create durante il taglio delle fratture.

3.13 cuttingfractures

La funzione divide una cella bidimensionale lungo una traccia all'interno di una mesh poligonale. Inizialmente vengono create due celle per contenere le parti risultanti della divisione, in seguito per ogni lato si verifica se vi sia un'intersezione con la traccia ed eventualmente vengono aggiornate le informazioni delle celle e della mesh. Le nuove celle vengono poi aggiunte alla lista *next* e la funzione restituisce *true* se la cella è stata divisa, *false* altrimenti. Sia n il numero di vertici della frattura, i due cicli sui vertici e sui lati hanno costo $O(n)$, così come la verifica per la presenza dell'intersezione.

3.14 checkIsNew

La funzione verifica se un punto è già un vertice di una cella bidimensionale nella mesh *PolygonalMesh*. Se il punto esiste restituisce *false* ed imposta l' *id* all'identificatore del vertice esistente, *true* altrimenti. Vi è un'iterazione sugli identificatori dei vertici e, per ogni vertice, vengono recuperate le coordinate del vertice corrispondente dalla mappa *MapCell0D* della mesh poligonale. Se le coordinate del vertice sono approssimativamente uguali a quelle del punto in input la funzione restituisce *false*. L'operazione col costo computazionale maggiore è iterare sui vertici del vettore *cd2*, detta n la dimensione di tale vettore il costo della funzione è dunque $O(n)$.

3.15 addNewVertAndEdg

Questa funzione aggiunge nuovi vertici e lati al poligono risultante dalla divisione di una *Cell2d*. Nel corpo della funzione vengono creati ed aggiornati i vertici e gli spigoli coinvolti nella divisione, operazioni elementari che portano ad un costo $O(1)$.

3.16 addNewEdg

La funzione aggiorna due nuove celle bidimensionali che si formano da un taglio aggiungendo nuovi spigoli e vertici in base ad un vertice comune *idSame* ed uno

preesistente *vert0*. Tutte le operazioni di aggiornamento richieste hanno costo costante, quindi la funzione ha una complessità computazionale $O(1)$.

3.17 dividingExistingVert

La funzione gestisce la divisione di una cella bidimensionale in due celle dello stesso tipo nel caso in cui il vertice in esame sia un vertice già esistente. Il corpo è costituito da operazioni ci confronto ed aggiornamento ed il costo è $O(1)$.

3.18 cuttedByNonPas

Questa funzione gestisce il caso in cui una *Cell2d* sia tagliata da una traccia non passante. La determinazione degli estremi della cella e la chiamata alla funzione *intersezionipoligonoretta* per trovare le intersezioni tra il poligono e la retta a cui la traccia appartiene portano ad un costo $O(N)$, dove N è il numero di vertici della cella.

3.19 splitOneEdg

La funzione gestisce la divisione di una cella bidimensionale *toCut* in due nuove *Cell2d* all'interno di una mesh poligonale. Vengono create delle nuove strutture per la memorizzazione dei dati aggiuntivi e si itera sugli spigoli della cella da tagliare determinando se siano "vecchi" o meno. Per quelli marcati come *old* vengono aggiunti i vertici ed i nuovi spigoli alla nuova cella, si aggiornano le liste degli identificatori e la nuova cella viene memorizzata come "toccata" dai nuovi spigoli, mentre per gli altri si procede soltanto all' aggiunta del vertice e dello spigolo della nuova cella. Infine la cella viene aggiunta alla mesh. Definito m come il numero di spigoli della cella *toCut*, il costo è $O(m)$.

3.20 printingPolygonMesh

La funzione ha lo scopo di stampare i dati delle mesh poligonali sul file di output. Vi è un ciclo esterno su tutte le mesh e, per ogni tipologia di cella, è presente un ciclo annidato in cui le informazioni caratteristiche di ogni tipo vengono estratte per essere stampate, inoltre per le *Cell2d* vi sono due cicli annidati aggiuntivi per la stampa delle *Cell0d* che le formano. Siano n il numero di oggetti di tipo *PolygonalMesh*, m_0, m_1 e m_2 il numero medio di *Cell0d*, *Cell1d* e *Cell2d* rispettivamente e sia infine v il numero medio di vertici per ogni *Cell2d*. Il costo totale della funzione è $O(n * (m_0 + m_1 + m_2 * v))$.

3.21 intersLato

La funzione permette di determinare se vi sia un'intersezione tra una traccia ed un lato, ricavato come differenza tra i suoi estremi. L'intersezione trovata risulta accettabile se è combinazione convessa degli estremi del lato e usando questa strategia si è evitata la risoluzione di un sistema lineare, operazione che

avrebbe avuto un costo computazionale maggiore. Il corpo è composto soltanto da operazioni aritmetiche e di confronto, pertanto il costo è $O(1)$.