AlphaShapes

Bevilacqua Luigi, Fakourfard Shahrzad

April 2021

Introduzione

Analisi e revisione del package AlphaShapes.jl

Obiettivi

- Studio del package AlphaStructures.jl e di tutte le funzioni e strutture dati utilizzate da esso.
- Descrivere per ogni funzione individuata (task), tipo e significato di ogni parametro e valore di ritorno.
- Suddivisione della funzione in singoli task.
- Stimare comportamenti e tempi delle singole funzioni puntando all'ottimizzazione delle stesse.
- Parallelizzare dove possibile

Analisi Preliminare

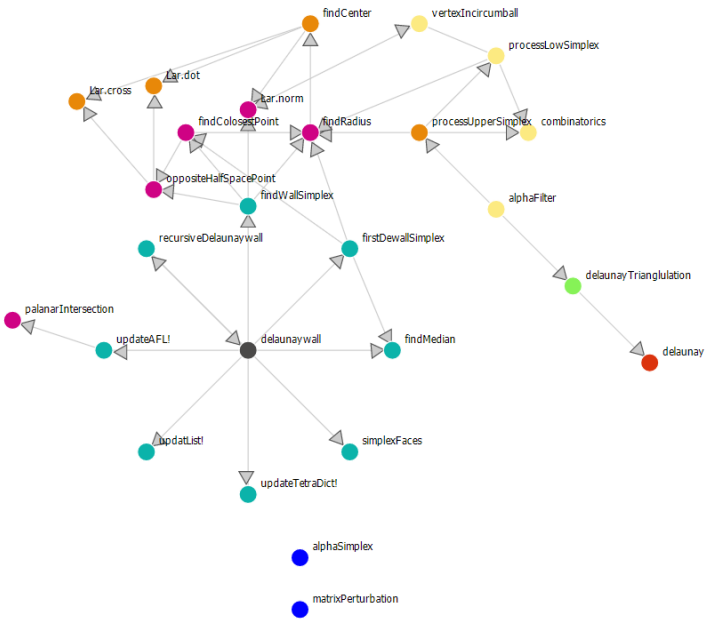


Figure 1: Le funzioni nel package AlphaStructures.jl e le relazioni tra loro

Analisi input e output delle funzioni

- **DelaunayWall**: prende in input una matrice MxN dove M è la dimensione dello spazio e N è numero di punti e restituisce come output Delaunay Triangulation dei punti dati.
- **findWallSimplex**: prende in input una matrice di punti, una face ed un punto specifico e restituisce come output un simplex che è un array.
- **firstDewallSimplex**: prende in input una matrice dei punti e restituisce la matrice degli indici dei punti dati che formano il primo tetraedro costruito sul Muro se l'asse "ax" ha il termine "off".
- **findMedian**: prende in input una matrice di punti e restituisce la mediana di punti rispetto all'asse "ax".
- **simplexFaces**: prende in input un simplex che è un array e restituisce in output le face possibili di quel simplex.
- **updateTetraDict!**: prende in input una matrice di punti, un simplex, una faccia e il dizionario chiamato tetraDict e aggiorna il contenuto del dizionario aggiungendogli i punti esterni della simplesso dato.
- **findAFL**: prende in input un array di punti e restituisce in output la lista delle faccie attivi dell'array dato.
- **updatList!**: prende in input la lista di facce ed una faccia e come l'output restituisce un valore booleano. Se una faccia esiste nella lista, non la considera e ritorna 'falso', se esiste ritorna 'vero' e la aggiunge alla lista.
- **updateAFL!**: prende in input una matrice di punti e le facce della matrice data, se una faccia non esiste nelle lista delle facce attive, gliela aggiunge,se esiste non la considera. l'output della funzione è un valore booleano che indica se il processo è finito con successo o meno.
- **recursiveDelaunaywall**: prende in input una matrice di punti,l'iperpiano con normale "ax" e il temine costante "ax". Restituisce la triangolazione di Delaunay per il sottospazio positivo o negativo della matrice data.
- **findColosestPoint**: prende in input una matrice di punti P ed una matrice Psimplex e restituisce l'indice del punto più vicino alla matrice Psimplex dalla matrice P.
- **findRadius**: prende in input una matrice di punti e restituisce il valore del raggio del circonferenza circoscritta ai punti dati.
- **oppositeHalfSpacePoint**:prende in input una matrice di punti e una face e restituisce in output gli indici dei punti che si trovano sulla face definita.

- **planarIntersection**: prende in input una matrice di punti, una faccia ed normall axis e restituisce in output 0 se la faccia si trova sulla iperpiano alpha, +1 se si trova sulla parte posistova dell'iperpiano alpha e -1 se si trova sulla parte negativo dell'iperpiano alpha.
- **findCenter**:prende in input una matrice di punti, restituisce il centro della circonferenza circoscritta ai punti dati.
- **processUpperSimplex**: prende in input una matrice di punti e la collezione ordinata di alpha filters e processa upper simplex.
- **processLowSimplex**: prende in input una matrice di punti e la collezione ordinata di alpha filters e processa low simplex.
- **vertexIncircumball**: prende in input una matrice di punti ed un raggio e restituisce in output un valore booleano se un punto si trova dentro del cerchio creato considerando il raggio definito o meno.
- **alphaFilter**: prende in input una matrice di punti e restituisce la collezione ordinata dei alpha filters.
- **delaunayTrianglulation**: prende una matrice di punti e restituisce un array dei simplessi di livello più alto della triangolazione di Delaunay.
- **alphaSimplex**: prende in input una matrice di punti e il dizionario dei filtri e un threshold e restituisce in output la collezione di d-simplex per d appartenente all'intervallo [0,dimension].
- **matrixPerturbation**:prende in input una matrice di punti e restetuisce una perturbazione per ogni valore della matrice. Questa funzione non viene chiamata da nessuna parte.

Refactoring

Il primo passo del refactoring del codice è stato suddividere le funzioni come segue:

- **findCenter()** è stata suddivisa in diverse funzioni in modo da poter utilizzare la macro @spawn per parallelizzare il codice (questo approccio però non ha dato i frutti sperati in quanto si peggioravano le prestazioni, quindi siamo tornati alla versione originale mantenendo solo il refactoring). La versione originale e modificata di questa funzione si trovano a `https://github.com/luigibvl/AlphaShapes.jl/blob/master/docs/geometry.ipynb`
- **delaunayWall()** è stata suddivisa in due funzioni chiamate computeFirstSimplex() e delaunayWall(). Le versione originale e modificata di questa funzione si trovano a `https://github.com/luigibvl/AlphaShapes.jl/blob/master/docs/deWall.ipynb`

– **recursiveDelaunayWall()** è stata suddivisa in due funzioni chiamate findAFL(), findTetraDict(). In questo modo abbiamo potuto utilizzare il macro @spawn per parallelizzare il codice. Le versione originale e modificata di questa funzione si trovano a `https://github.com/luigibvl/AlphaShapes.jl/blob/master/docs/deWall.ipynb`

Parallelizzazione

Nel secondo passo abbiamo trasformato le list comprehension in for loop trazionionali, per poter utilizzare le macro, ma dopo la trasformazione del codice e comparando i tempi di esecuzione abbiamo scoperto che le list comprehension sono generalmente più veloci rispetto ai cicli for tradizionali con le macro, quindi in questo caso abbiamo deciso di utilizzare il codice originale. Quando invece le prestazioni miglioravano abbiamo utilizzato la macro **@simd** e **@inbounds** (la macro **@inbounds** serve a dare al complilatore una direttiva per rimuovere il **@boundscheck**, quindi questa macro va utilizzata con attenzione).

Per parallelizzare il codice ed aumentare la velocità di esecuzione del codice, abbiamo utilizzato diverse macro native di Julia:

– **@spawn**: come sappiamo questa macro è il modo semplice per eseguire una funzione in un processo remoto senza dover specificare il nodo remoto.

Esempio dell'utilizzo di questa macro in questo progetto è il seguente:

```
AFL = @spawn AlphaStructures.simplexFaces(σ)
AFL = fetch(AFL)
```

– **@sync**: come sappiamo per parallelizzare un ciclo for quando vogliamo che il ciclo attenda ed osservi l'esecuzione delle attività avviate al suo interno, usiamo la macro @sync.

Esempio dell'utilizzo di questa macro in questo progetto è il sequente:

```
@sync for (k, v) in filtration
        if v <= α_threshold
            push!(simplexCollection[length(k)], k)
        end
    end
```

– **@simd**: un'altra macro per parallelizzare i cicli for è @simd. Utilizzo di questo macro su un ciclo for dà al compilatore la libertà di usare le istruzioni SIMD per le operazioni all'interno del ciclo, se è possibile.

Esempio dell'utilizzo di questa macro in questo progetto è il seguente:

```
@simd for face in newσ
    inters = AlphaStructures.planarIntersection(P, face, ax, off)
```

```
    if inters == 0 # intersected by plane α
        AlphaStructures.updatelist!(AFLα, face)
    elseif inters == -1 # in NegHalfspace(α)
        AlphaStructures.updatelist!(AFLminus, face)
    elseif inters == 1 # in PosHalfspace(α)
        AlphaStructures.updatelist!(AFLplus, face)
    else
        return false
    end
end
```

– **@threads**: questa macro viene utilizzata per parallelizzare un ciclo for da eseguire con più thread. Divide lo spazio di iterazione tra più attività ed esegue tali attività sui thread in base a una politica di pianificazione. Alla fine del ciclo viene posta una barriera che attende il completamento dell'esecuzione di tutte le attività.

Esempio dell'utilizzo di questa macro in questo progetto è il sequente:

```
@threads for cell in AFL
        point = setdiff(σ, cell)
        @assert length(point) == 1 "updateTetraDict!: Error during"
                                    "update of TetraDict $σ, $cell"
        tetraDict[ cell ] = P[:, point[1]]
    end
```

Conclusioni

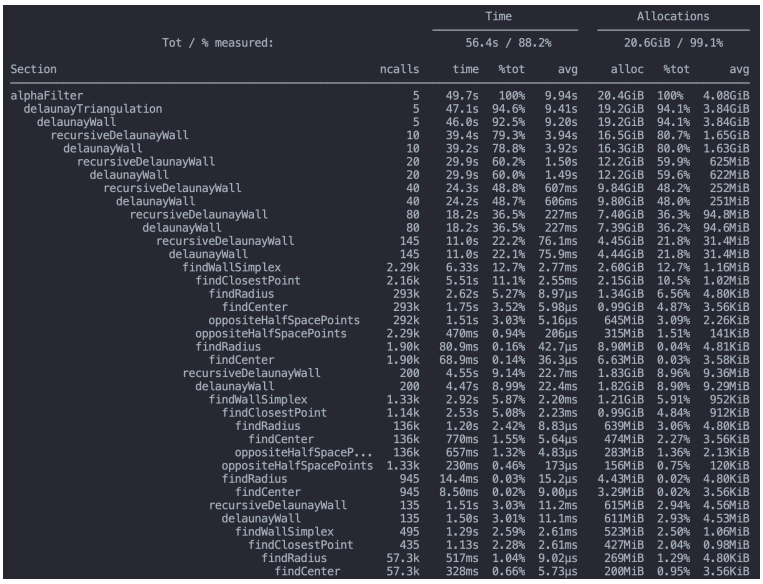
Di seguito vengono mostrati alcuni esempi di applicazione della macro **@timeit** afferente al package TimerOutput, utilizzato per generare output formattato. Esempio nel caso 2D e 3D (file utilizzato in 2D: Lar2.svg, file utilizzato in 3D: teapot.obj. Nel caso dell'esempio 3D abbiamo tagliato parte dell'immagine poichè troppo grande)

		Time			Allocations		
Tot / % measured:		6.92s / 1.84%			2.09GiB / 1.40%		
Section	ncalls	time	%tot	avg	alloc	%tot	avg
alphaFilter	1	68.7ms	54.1%	68.7ms	26.2MiB	87.3%	26.2MiB
processuppersimplex	1.98k	66.2ms	52.1%	33.5µs	25.0MiB	83.3%	13.0KiB
processlowsimplex	5.92k	32.7ms	25.7%	5.52µs	15.1MiB	50.3%	2.61KiB
findRadius	5.92k	10.3ms	8.13%	1.74µs	5.97MiB	19.9%	1.03KiB
findCenter	5.92k	1.98ms	1.56%	335ns	2.17MiB	7.24%	384B
vertexInCircumball	5.92k	5.81ms	4.58%	981ns	2.89MiB	9.65%	512B
findCenter	5.92k	1.91ms	1.50%	322ns	2.17MiB	7.24%	384B
findRadius	1.98k	27.8ms	21.9%	14.1µs	7.77MiB	25.9%	4.03KiB
findCenter	1.98k	6.99ms	5.50%	3.54µs	5.39MiB	18.0%	2.80KiB
delaunayTriangulation	1	1.28ms	1.01%	1.28ms	676KiB	2.20%	676KiB
alphaSimplex	38	58.3ms	45.9%	1.53ms	3.80MiB	12.7%	102KiB

Figure 2: caso 2D col modulo originale AlphaStructures

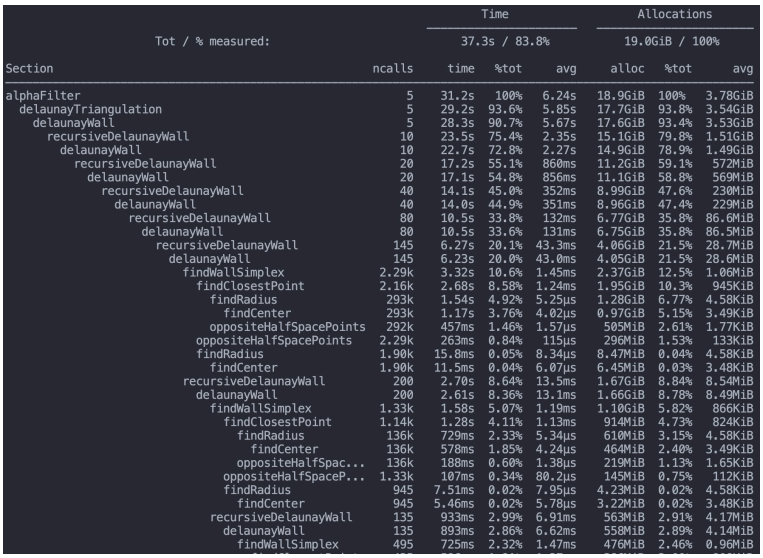
		Time			Allocations		
Tot / % measured:		5.59s / 4.41%			1.93GiB / 2.07%		
Section	ncalls	time	%tot	avg	alloc	%tot	avg
alphaFilter	1	204ms	82.6%	204ms	37.0MiB	90.6%	37.0MiB
delaunayTriangulation	1	60.2ms	24.4%	60.2ms	2.14MiB	5.24%	2.14MiB
processuppersimplex	1.98k	48.9ms	19.9%	24.8µs	23.0MiB	56.3%	11.9KiB
processlowsimplex	5.92k	38.9ms	15.8%	6.56µs	14.3MiB	35.0%	2.47KiB
findRadius	5.92k	23.9ms	9.69%	4.03µs	5.33MiB	13.1%	944B
findCenter	5.92k	1.83ms	0.74%	310ns	2.17MiB	5.32%	384B
vertexInCircumball	5.92k	3.61ms	1.47%	610ns	2.71MiB	6.65%	480B
findCenter	5.92k	1.78ms	0.72%	301ns	2.17MiB	5.32%	384B
findRadius	1.98k	5.81ms	2.36%	2.94µs	6.57MiB	16.1%	3.41KiB
findCenter	1.98k	4.21ms	1.71%	2.13µs	5.12MiB	12.6%	2.66KiB
alphaSimplex	38	42.9ms	17.4%	1.13ms	3.82MiB	9.37%	103KiB

Figure 3: caso 2D col nuovo modulo AlphaStructures



Tot / % measured:		Time			Allocations		
		56.4s / 88.2%			20.6GiB / 99.1%		
Section	ncalls	time	%tot	avg	alloc	%tot	avg
alphaFilter	5	49.7s	100%	9.94s	20.4GiB	100%	4.08GiB
delaunayTriangulation	5	47.1s	94.6%	9.41s	19.2GiB	94.1%	3.84GiB
delaunayWall	5	46.0s	92.5%	9.20s	19.2GiB	94.1%	3.84GiB
recursiveDelaunayWall	10	39.4s	79.3%	3.94s	16.5GiB	80.7%	1.65GiB
delaunayWall	10	39.2s	78.8%	3.92s	16.3GiB	80.0%	1.63GiB
recursiveDelaunayWall	20	29.9s	60.2%	1.50s	12.2GiB	59.9%	625MiB
delaunayWall	20	29.9s	60.0%	1.49s	12.2GiB	59.6%	622MiB
recursiveDelaunayWall	40	24.3s	48.8%	607ms	9.84GiB	48.2%	252MiB
delaunayWall	40	24.2s	48.7%	606ms	9.80GiB	48.0%	251MiB
recursiveDelaunayWall	80	18.2s	36.5%	227ms	7.40GiB	36.3%	94.8MiB
delaunayWall	80	18.2s	36.5%	227ms	7.39GiB	36.2%	94.6MiB
recursiveDelaunayWall	145	11.0s	22.2%	76.1ms	4.45GiB	21.8%	31.4MiB
delaunayWall	145	11.0s	22.1%	75.9ms	4.44GiB	21.8%	31.4MiB
findWallSimplex	2.29k	6.33s	12.7%	2.77ms	2.60GiB	12.7%	1.16MiB
findClosestPoint	2.16k	5.51s	11.1%	2.55ms	2.15GiB	10.5%	1.02MiB
findRadius	293k	2.62s	5.27%	8.97µs	1.34GiB	6.56%	4.80KiB
findCenter	293k	1.75s	3.52%	5.98µs	0.99GiB	4.87%	3.56KiB
oppositeHalfSpacePoints	292k	1.51s	3.03%	5.16µs	645MiB	3.09%	2.26KiB
oppositeHalfSpacePoints	2.29k	470ms	0.94%	206µs	315MiB	1.51%	141KiB
findRadius	1.90k	80.9ms	0.16%	42.7µs	8.90MiB	0.04%	4.81KiB
findCenter	1.90k	68.9ms	0.14%	36.3µs	6.63MiB	0.03%	3.58KiB
recursiveDelaunayWall	200	4.55s	9.14%	22.7ms	1.83GiB	8.96%	9.36MiB
delaunayWall	200	4.47s	8.99%	22.4ms	1.82GiB	8.90%	9.29MiB
findWallSimplex	1.33k	2.92s	5.87%	2.20ms	1.21GiB	5.91%	952KiB
findClosestPoint	1.14k	2.53s	5.08%	2.23ms	0.99GiB	4.84%	912KiB
findRadius	136k	1.20s	2.42%	8.83µs	639MiB	3.06%	4.80KiB
findCenter	136k	770ms	1.55%	5.64µs	474MiB	2.27%	3.56KiB
oppositeHalfSpaceP...	136k	657ms	1.32%	4.83µs	283MiB	1.36%	2.13KiB
oppositeHalfSpacePoints	1.33k	230ms	0.46%	173µs	156MiB	0.75%	120KiB
findRadius	945	14.4ms	0.03%	15.2µs	4.43MiB	0.02%	4.80KiB
findCenter	945	8.50ms	0.02%	9.00µs	3.29MiB	0.02%	3.56KiB
recursiveDelaunayWall	135	1.51s	3.03%	11.2ms	615MiB	2.94%	4.56MiB
delaunayWall	135	1.50s	3.01%	11.1ms	611MiB	2.93%	4.53MiB
findWallSimplex	495	1.29s	2.59%	2.61ms	523MiB	2.50%	1.06MiB
findClosestPoint	435	1.13s	2.28%	2.61ms	427MiB	2.04%	0.98MiB
findRadius	57.3k	517ms	1.04%	9.02µs	269MiB	1.29%	4.80KiB
findCenter	57.3k	328ms	0.66%	5.73µs	200MiB	0.95%	3.56KiB

Figure 4: caso 3D col modulo originale AlphaStructures



Tot / % measured:		Time			Allocations		
		37.3s / 83.8%			19.0GiB / 100%		
Section	ncalls	time	%tot	avg	alloc	%tot	avg
alphaFilter	5	31.2s	100%	6.24s	18.9GiB	100%	3.78GiB
delaunayTriangulation	5	29.2s	93.6%	5.85s	17.7GiB	93.8%	3.54GiB
delaunayWall	5	28.3s	90.7%	5.67s	17.6GiB	93.4%	3.53GiB
recursiveDelaunayWall	10	23.5s	75.4%	2.35s	15.1GiB	79.8%	1.51GiB
delaunayWall	10	22.7s	72.8%	2.27s	14.9GiB	78.9%	1.49GiB
recursiveDelaunayWall	20	17.2s	55.1%	860ms	11.2GiB	59.1%	572MiB
delaunayWall	20	17.1s	54.8%	856ms	11.1GiB	58.8%	569MiB
recursiveDelaunayWall	40	14.1s	45.0%	352ms	8.99GiB	47.6%	230MiB
delaunayWall	40	14.0s	44.9%	351ms	8.96GiB	47.4%	229MiB
recursiveDelaunayWall	80	10.5s	33.8%	132ms	6.77GiB	35.8%	86.6MiB
delaunayWall	80	10.5s	33.6%	131ms	6.75GiB	35.8%	86.5MiB
recursiveDelaunayWall	145	6.27s	20.1%	43.3ms	4.06GiB	21.5%	28.7MiB
delaunayWall	145	6.23s	20.0%	43.0ms	4.05GiB	21.5%	28.6MiB
findWallSimplex	2.29k	3.32s	10.6%	1.45ms	2.37GiB	12.5%	1.06MiB
findClosestPoint	2.16k	2.68s	8.58%	1.24ms	1.95GiB	10.3%	945KiB
findRadius	293k	1.54s	4.92%	5.25µs	1.28GiB	6.77%	4.58KiB
findCenter	293k	1.17s	3.76%	4.02µs	0.97GiB	5.15%	3.49KiB
oppositeHalfSpacePoints	292k	457ms	1.46%	1.57µs	505MiB	2.61%	1.77KiB
oppositeHalfSpacePoints	2.29k	263ms	0.84%	115µs	296MiB	1.53%	133KiB
findRadius	1.90k	15.8ms	0.05%	8.34µs	8.47MiB	0.04%	4.58KiB
findCenter	1.90k	11.5ms	0.04%	6.07µs	6.45MiB	0.03%	3.48KiB
recursiveDelaunayWall	200	2.70s	8.64%	13.5ms	1.67GiB	8.84%	8.54MiB
delaunayWall	200	2.61s	8.36%	13.1ms	1.66GiB	8.78%	8.49MiB
findWallSimplex	1.33k	1.58s	5.07%	1.19ms	1.10GiB	5.82%	866KiB
findClosestPoint	1.14k	1.28s	4.11%	1.13ms	914MiB	4.73%	824KiB
findRadius	136k	729ms	2.33%	5.34µs	610MiB	3.15%	4.58KiB
findCenter	136k	578ms	1.85%	4.24µs	464MiB	2.40%	3.49KiB
oppositeHalfSpac...	136k	188ms	0.60%	1.38µs	219MiB	1.13%	1.65KiB
oppositeHalfSpaceP...	1.33k	107ms	0.34%	80.2µs	145MiB	0.75%	112KiB
findRadius	945	7.51ms	0.02%	7.95µs	4.23MiB	0.02%	4.58KiB
findCenter	945	5.46ms	0.02%	5.78µs	3.22MiB	0.02%	3.48KiB
recursiveDelaunayWall	135	933ms	2.99%	6.91ms	563MiB	2.91%	4.17MiB
delaunayWall	135	893ms	2.86%	6.62ms	558MiB	2.89%	4.14MiB
findWallSimplex	495	725ms	2.32%	1.47ms	476MiB	2.46%	0.96MiB

Figure 5: caso 3D col nuovo modulo AlphaStructures