UNIVERSITÀ DEGLI STUDI
DI PARMA

DIPARTIMENTO DI MATEMATICA E INFORMATICA
Corso di Laurea in: INFORMATICA

Studio di algoritmi per la rappresentazione di dati geografici con multi risoluzione

Studente:
Pietralberto Mazza

Relatore:
Alessandro Dal Palù

Anno Accademico 2015/2016

Alla famiglia e ai BdM

Indice

Elenco delle figure

Introduzione

Negli utimi anni in Europa sono sempre maggiori i danni legati ad eventi di inondazione e nonostante sia stato stimato che nel 2050 la popolazione colpita da queste calamità varierà tra le 500.000 e le 640.000 unità, con un impatto economico tra i 20-40 mld di euro, sono ancora più allarmanti i dati derivanti dalle proiezioni del 2080 che mostrano un aumento di vittime (540.000-950.000 all'anno) e un conseguente aumento del derivante danno economico (30-100 mld di euro).
Dopo le nuove Direttive Europee, molti paesi hanno istituito programmi per l'analisi e la valutazione del rischio idrogeologico e in questo tipo di programmi la modellazione e la simulazione numerica dei flussi d'acqua rimane uno degli strumenti principalmente utilizzati.

Il dipartimento di Matematica e Informatica in collaborazione col dipartimento di Ingegneria Civile e Ambientale e Architettura ha sviluppato un programma per la simulazione dei flussi d'acqua su aree di interesse variabili. L'applicativo, come è facile pensare, richiede in input mappe geografiche di grandi dimensioni (di intere province ad esempio) e, data la grossa mole di informazione che queste mappe contenogono, è stato necessario introdurre nuove strutture dati in grado di rappresentarle in multi risoluzione.

Questa tesi si focalizza sulla fase di pre-proccessing in cui, data una mappa geografica di grandi dimensioni, viene eseguito un processo che restituisce l'immagine stessa in multi risoluzione, rendendo così possibile la valutazione di effeti su scala ridotta rispetto al grande dominio dell'intera immagine.
In particolare bisogna precisare che questa fase era già stata implementata e che mi sono occupato di modificarla per renderla più efficiente.

È necessario introdurre anche che il programma che compie le simulazioni di corsi d'acqua è implementato con codice che sfrutta la parallelizzazione del calcolo in più unità di elaborazione che risiedono in componenti specifici chiamati GPU (Graphic Processing Unit).

Il capitolo 1 di questo documento contiene le nozioni preliminari utili

a comprendere la struttura e il funzionamento del progetto. Il capitolo 2 descrive brevemente lo stato dell'arte, l'obiettivo della tesi e il nuovo approccio, mentre nel capitolo 3 vengono illustrate nello specifico le nuove soluzioni adottate e come queste portano ad una maggiore efficienza. Infine nel capitolo 4 sono riportati i risultati e alcuni test cases e nel capitolo 5 le conclusioni e i possibili sviluppi futuri.

Capitolo 1

Nozioni preliminari

1.1 Immagini

Un'immagine in computer grafica viene rappresentata come una matrice rettangolare di punti chiamati pixels. In questo progetto, utilizzando mappe geografiche altimetriche, ogni punto della matrice rappresenta, invece, l'altezza del terreno di una area specifica, quindi oltre all'altezza vengono memorizzati anche la larghezza del singolo pixel (1m ad esempio) e la georeferenziazione, ovvero si specifica la posizione geografica di un punto della matrice, in modo da poter posizionare la matrice correttamente nello spazio.

1.1.1 GIS

Le mappe geografiche utilizzate in questo progetto vengono prese da un particolare database geografico chiamato GIS. Un Geographic Information System è un sistema progettato per ricevere, immagazzinare, elaborare, analizzare, gestire e rappresentare dati di tipo geografico.
Le immagini utilizzate in questo lavoro di tesi sono una rielaborazione compiuta dal dipartimento di Ingegneria Idraulica sulla base delle immagini GIS.

1.1.2 Tavolette

Per semplificare la gestione di queste grandi immagini, va tuttavia precisato che una generica mappa è in realtà composta da più sottomatrici chiamate tavolette e, quindi, puo essere vista come una matrice N x M di tavolette, che a loro volta sono matrici n x m di punti. Nel lavoro di tesi

si assume che ogni immagine sia composta da una griglia ordinata di file numerati da 1 a $N * M$ con estensione .grd.
Un file .grd contiene un'intestazione in cui sono indicati:

- il numero di righe e di colonne della tavoletta (*n*,*m*)
- il punto con coordinate minime e il punto con coordinate massime (il vertice in basso a sinistra e il vertice in alto a destra della tavoletta)
- la coppia di altezze del terreno con valori minimo e massimo

```
DSAA
631 301
642000 642630
4947500 4947800
44.64 49.2
48.89 48.88 48.88 48.94 48.94
48.95 49 49.02 49.07 49.07
[...]
```

Figura 1.1: Header di un file .grd

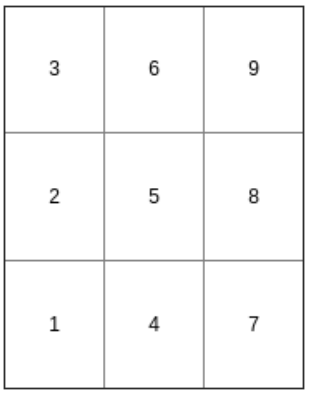


Figura 1.2: Ordinamento delle tavolette nella matrice

Quello che segue dopo l'header sono le altezze del terreno per ogni punto della tavoletta.
Ogni file viene dunque letto e salvato in un array di tavolette in cui sono memorizzate solo le informazioni necessarie presenti nell'header:

- la dimensione di ogni tavoletta (numero di righe e di colonne)
- le coordinate dei punti min e max

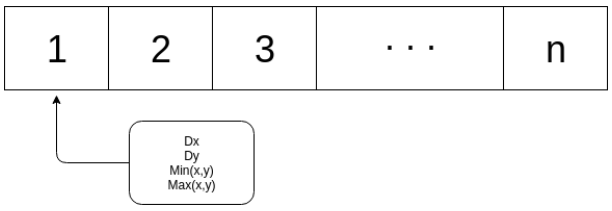


Figura 1.3: Array di tavolette

1.1.3 Poligono

In genere vengono utilizzate immagini di mappe geografiche e data la loro grandezza, sulla mappa viene delineata un'area di interesse, ovvero un poligono che delimita la zona entro la quale compiere una simulazione di un flusso d'acqua. Questo poligono viene salvato in un file con estensione .BLN, all'interno del quale vengono scritte le coordinate geografiche x,y dei vertici, seguendo il senso orario, oltreché il tipo di condizione di bordo:

- 0 - Nessuna condizione
- 1 - Muro
- 2 - Acqua in entrata
- 3 - Acqua in uscita
- 4 - Terreno lontano

Le informazioni del poligono vengono salvate in una struttura dati composta (`struct`) contenente 2 array, rispettivamente quello delle coordinate dei vertici e quello delle condizioni di bordo.

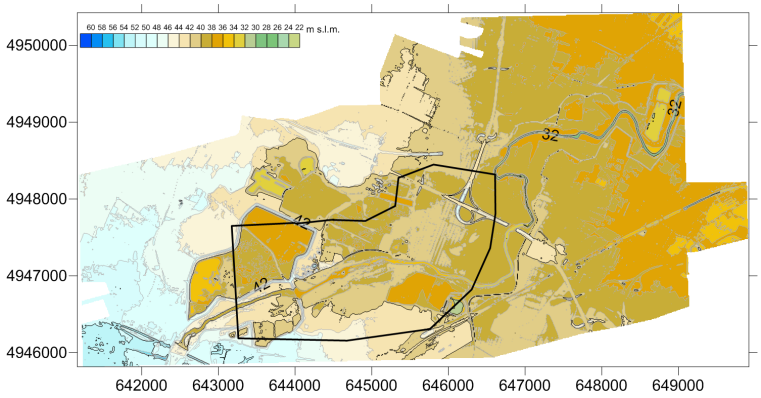


Figura 1.4: Poligono in una mappa altimetrica

1.2 Multi risoluzione

Quando un'immagine viene discretizzata con una griglia cartesiana, i dati sono memorizzati con un array bidimensionale. Date, dunque, le coordinate del vertice minimo della griglia, gli indici della matrice utilizzati per scorrere l'array permettono di calcolare le coordinate reali usate nel sistema di riferimento attraverso la formula:

```
real_x = min_x + j * dx;
real_y = min_y + i * dy
```

Una matrice a multi risoluzione è una matrice di 10-50 volte ridotta rispetto alle dimensioni dell'array bidimensionale, ed è composta da blocchi a differenti livelli di risoluzione.

1.2.1 Blocco

Ogni blocco contiene *BS* x *BS* celle ma vengono usati diversi livelli di risoluzione:

- livello 1 con cella di dimensione Δ_1
- livello 2 con cella di dimensione $\Delta_2 = 2 * \Delta_1$

  $\vdots$
- livello n con cella di dimensione $\Delta_n = 2^{n-1} * \Delta_1$

Ogni cella rappresenta un punto in coordinate geografiche. A livello 1, ovvero a risoluzione massima, ogni cella rappresenta esattamente un punto geografico (Δ_1=distanza fra i puni geografici), mentre a livelli di risoluzione più alti, ovvero a risoluzione minore, le celle rappresentano punti geografici a distanza Δ_i. In questa tesi *BS* viene assunto uguale a 8 o a 16, ma può avere valore uguale a qualunque potenza di 2, con n = 4 livelli di risoluzione.
Nella matrice a multi risoluzione il valore di una cella di un blocco a massima risoluzione è uguale all'altezza del terreno del rispettivo punto in coordinate geografiche, mentre il valore di una a risoluzione minore rappresenta una media delle altezze di $\Delta_i * \Delta_i$ punti geografici.

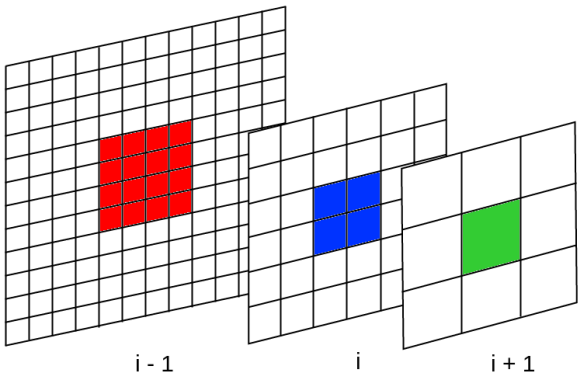


Figura 1.5: Relazione fra i blocchi di diverso livello

1.2.2 Matrice di blocchi

Una volta creati, i blocchi vengono codificati e memorizzati in una matrice in cui sono ordinati secondo il numero di blocco.
Così facendo viene modificato l'originale rapporto di vicinanza fra i vari blocchi; in seguito verrà mostrato come, con opportuni calcoli e opportune strutture dati di supporto, è possibile mantenerlo.

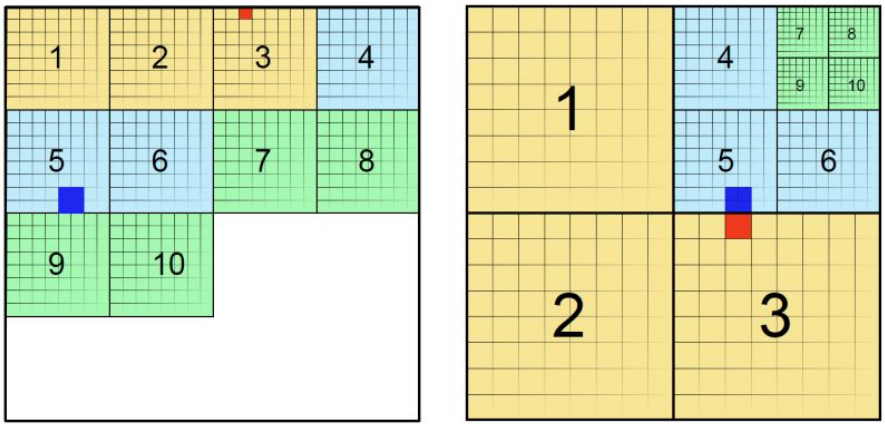


Figura 1.6: Esempio di matrice a multi risoluzione (a sinistra) e di matrice dei blocchi (a destra)

Capitolo 2

Obiettivi

2.1 Stato dell'arte

Il codice originale di partenza è strutturato in maniera tale che in una prima fase vengano letti i dati relativi all'immagine e al poligono e salvati rispettivamente in un array bidimensionale e in una struct contentente due array mono dimensionali (vertici e condizioni di bordo).

Viene, poi, eseguito un controllo per ogni punto della matrice per stabilire se questo sia interno o esterno al poligono e, in seguito, per tutte le celle vengono calcolate le condizioni di bordo.
Una volta terminate le fasi di caricamento e di calcolo delle informazioni necessarie, viene generata la multi risoluzione sull'intera mappa a partire da un set di seed points letti in input da file.

Il problema legato a questo tipo di approccio sta nel fatto che la matrice a multi risoluzione viene creata sulla base delle dimensioni della matrice che rappresenta la mappa altimetrica.
Anche se le dimensioni della matrice a multi risoluzione sono inferiori fra le 10 e le 50 volte rispetto a quelle dell'immagine, la memoria da allocare rappresenta un collo di bottiglia per il programma che, in seguito, sfrutta la matrice a multi risoluzione per la simulazione del corso d'acqua.

2.2 Obiettivo della tesi

L'obiettivo di questa tesi è, dunque, quello di modificare il codice sorgente adottanto un approccio diverso in modo tale da creare la matrice a multi risoluzione prima del caricamento delle tavolette, e progettare l'algoritmo così da ìoccupare meno spazio in memoria per la memorizzazione della matrice a multi risoluzione.

2.3 Soluzioni adottate

La soluzione adottata per ovviare al problema dell'allocazione di memoria è realizzata così da caricare inizialmente la struttura della matrice di tavolette e gli array del poligono e individuare la sottomatrice (bounging box) in cui questo ricade, sfruttando gli indici della matrice, calcolabili attraverso le coordinate geografiche (vengono lette solo le intestazioni delle tavolette!).
Operando sul bounding box e partendo, poi, da un insieme di punti che rappresentano il perimetro del poligono, viene creata la matrice a multi risoluzione, caricando le altezze di tutti i punti delle tavolette solo alla fine, riducendo così lo spazio di memoria d'allocazione inizializzando, in una prima fase iniziale, tutti i valori delle celle della matrice ad un valore di default. Quando, in seguito, vengono lette le informazioni delle tavolette, il problema a cui bisogna ovviare è quello di caricare i valori delle altezze del terreno nelle corrette celle della matrice a multi risoluzione, che viene risolto attraverso opportuni calcoli ed opportune strutture dati di supporto.

Capitolo 3

Realizzazione

Il progetto è stato implementato con il linguaggio di programmazione C++ ed è suddiviso in 2 parti: bln_reader, la prima parte relativa all'elaborazione della matrice di tavolette e del poligono e multires, la seconda fase in cui, sfruttando le informazioni precedentemente acquisite, viene creata la matrice a multi risoluzione.

3.1 Bln_reader

```
int main() {

    string BLN_path = "polygons/bln_secchia";
    string GRD_path = "slabs/";

    read_bln(BLN_path);

    int n_slab = count_grd(GRD_path);

    read_grd(GRD_path, n_slab);

    bounding_box(n_slab);

    bln_interpolation();

    return 0;

}
```

3.1.1 Lettura dell'input

Con la funzione `read_bln()` vengono lette le informazioni contenute nel file relativo al poligono disegnato sulla mappa geografica, che vengono memorizzate in una `struct` di tipo polygon:

```
typedef struct point_ {
    double x,y;
} point;

typedef struct polygon_ {
    vector<point> points;
    vector<int> edges;
} polygon;

polygon pol;
```

I vertici del poligono sono elencati nel file .BLN seguendo il senso orario e vengono espressi sotto forma di coordinate cartesiane x,y che vengono salvate nell'array `pol.points`. In `pol.edges` sono memorizzate le condizioni di bordo di ogni lato del poligono.
In `read_bln()` vengono inoltre memorizzati nelle variabili `pol_mp` e `pol_Mp` i valori minimi e massimi di `pol.points[i].x` e `pol.points[i].y`, così da individuare il vertice in basso a sinistra e quello in alto a destra del rettangonolo che inscrive il poligono.

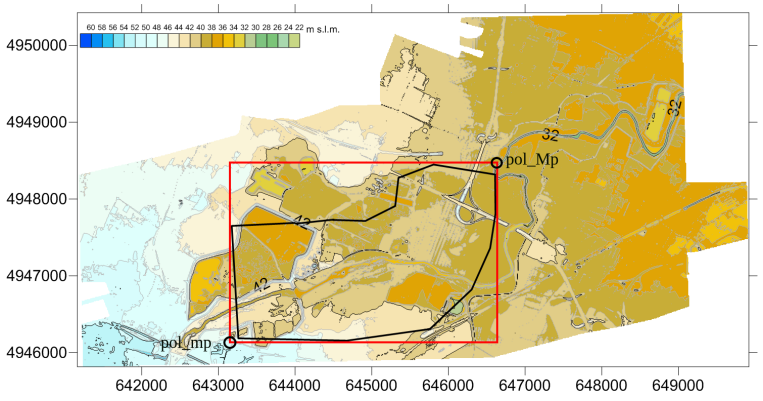


Figura 3.1: Il rettangolo che inscrive il poligono

I file delle tavolette sono collocati nella directory `slabs/` e sono numerati da 1 a n.

Attraverso la funzione `count_grd()` viene eseguito un bash script che restituisce il numero di file nella cartella, in modo tale da poterli leggere sfruttando un ciclo `for`.

Il metodo `read_grd()` legge le informazioni contenute negli header delle tavolette, che vengono salvate in una struct di tipo slab:

```
typedef struct slab_ {
    point m_points;
    point M_points;
    int dx;
    int dy;
}slab;

vector<slab> slabs;
```

In ogni elemento dell'array `slabs` sono memorizzate le informazioni della rispettiva tavoletta, quali le coordinate (*x*,*y*) del punto con valore minore, ovvero il vertice in basso a sinistra, quelle del vertice in alto a destra, cioè le coordinate del punto con valore maggiore e le dimensioni della tavoletta (*n_colonne* x *n_righe*).

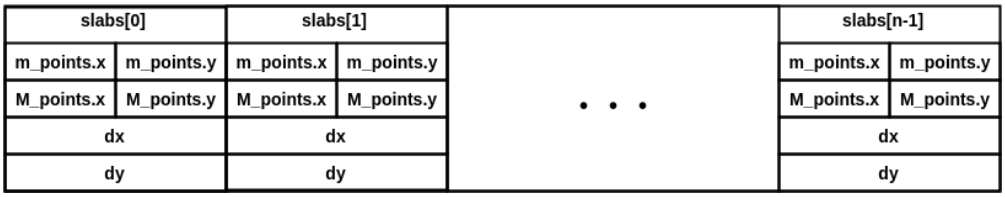


Figura 3.2: Vettore di tavolette

Anche in `read_grd()` vengono memorizzate informazioni aggiuntive riguardo la griglia cartesiana:
due variabili chiamate `s_min` ed `s_max` contengono i valori delle coordinate minime e massime della matrice di tavolette. Sfruttando, ora, i dati contenuti in `pol_mp` ed `s_min` è possibile calcolare i valori delle coordinate minime della tavoletta che contiene `pol_mp`, ovvero il vertice in basso a sinistra del rettangolo che inscrive il poligono. Queste coordinate sono salvate in una variabile chiamata `min_xy`.

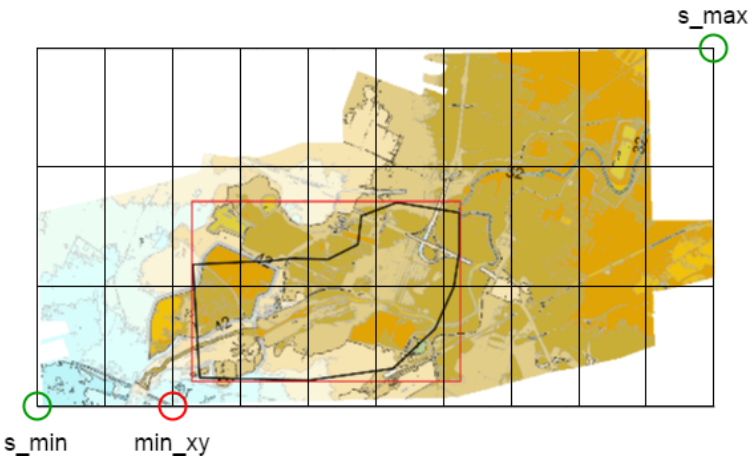


Figura 3.3: Mappa geografica suddivisa in tavolette

3.1.2 Bounding Box

Il numero di righe e di colonne della matrice di tavolette è calcolabile attraverso la formula:

```
cols = (s_max.x - s_min.x) / slabs[0].dx + 1;
rows = (s_max.y - s_min.y) / slabs[0].dy + 1;
```

Calcolando la distanza sull'asse x tra i vertici della matrice di tavolette, dividendola per il numero di colonne contenute in una singola tavoletta e sommando uno si ottiene quindi il numero di colonne totali della matrice. In modo analogo si calcolano il numero delle righe e gli indici i,j di una qualsiasi tavoletta.
La sottomatrice di tavolette che comprende l'intero rettangolo è chiamata **bounding box** e per individuarla basta trovare gli indici i,j delle tavolette che contengono i vertici in basso a sinistra e in alto a destra del rettangolo:

```
//indici matriciali della tavoletta che contiene il vertice
//in basso a sinistra del rettangolo
x = (pol_mp.y - s_min.y) / slabs[0].dy;
y = (pol_mp.x - s_min.x) / slabs[0].dx;
```

Dato l'ordine specifico in cui sono disposte le tavolette all'interno della matrice (fig 1.2), la tavoletta con coordinate `s_min`, ovvero quella che si assume essere lo 0, è in realtà in posizione (`rows-1`,0), dunque l'indice i di tutte le tavolette viene calcolato secondo la formula:

```
i = (rows - 1) - x;
```

Le informazioni relative al bounding box vengono scritte nel file `map_info.txt` assieme ad altri dati necessari in `multires()` al momento del caricamento dei valori delle altezze di ogni punto delle tavolette.

3.1.3 Perimetro del poligono

Attraverso la funzione `bln_interpolation()` viene eseguita l'interpolazione del perimetro del poligono, cioè ogni segmento che congiunge i vertici viene espresso come una successione di punti.
Per trovare questi punti, per ogni coppia di vertici viene calcolata la differenza fra le rispettive coordinate x e coordinate y (l'ultima coppia è formata dall'ultimo e dal primo vertice) ed utilizzando il teorema di Pitagora viene calcolata la lunghezza del segmento che congiunge i due punti:

```
//npoints numero di vertici del poligono
x0 = pol.points[i].x;
y0 = pol.points[i].y;
x1 = pol.points[(i+1) % npoints].x;
y1 = pol.points[(i+1) % npoints].y;

deltax = x1 - x0;
deltay = y1 - y0;

dd = sqrt(deltax * deltax + deltay * deltay);
```

Per calcolare le coordinate del primo punto che sta sul segmento `dd` basta sommare le quantità:

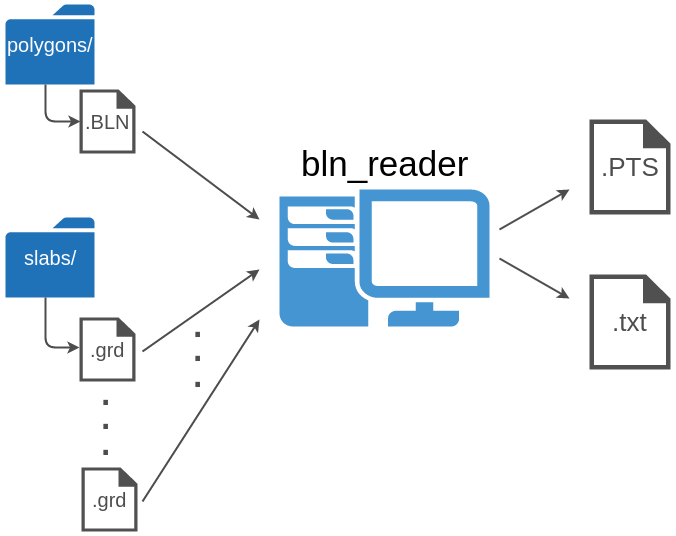
```
deltax / dd;
deltay / dd;
```

alle coordinate di un vertice. Per calcolare, invece, le coordinate di tutti gli altri punti basta sostituire le coordinate del nuovo punto ottenuto a quelle del vertice nelle operazioni.
Con questa procedura, per ogni coppia di vertici si ottiene una lista di punti equispaziati tra loro.

3.1.4 Output del programma

Ogni nuovo punto che viene calcolato viene trascritto su un file chiamato `bln_raster.PTS`. Al termine delle operazioni, questo file, assieme a `map_info.txt` rappresentano l'effettivo output del programma.

`Bln_reader`, dunque, dati in input una lista di file .grd rappresentanti un'immagine e un file .BLN contenente la lista delle coordinate dei vertici di un poligono disegnato sopra questa mappa, restituisce in output altri due file, un `.txt` in cui sono elencate informazioni necessarie per creare la matrice a multi risoluzione, e un .PTS formato dalla lista di punti che compongono il perimetro del poligono.



3.2 Multires

```
int main() {

    string map_file = "map_info.txt";
    string bln_file = "polygons/bln_secchia";
    string pts_file = "polygons/bln_raster.PTS";
    string path = "slabs/";

    read_map(map_file);

    read_pts(pts_file);

    read_bln(bln_file);

    multires(path);

    return 0;

}
```

3.2.1 Input

`Multires` si serve di una `struct` chiamata `maps` in cui vengono salvate molte informazioni tra cui quelle passate in `bln_reader` attraverso il file `map_info.txt`. È importante ricordare che `multires` opera sul bounding box e non sull'intera mappa e, attraverso `read_map()`, viene letto in primo luogo il contenuto di `map_info.txt` che caratterizza questa sotto matrice ed è composto da:

- il numero di righe totali della matrice di tavolette
- la dimensione del bounding box (nrighe ed ncolonne di punti!)
- le coordinate del punto `min_xy`
- il passo (la distanza) fra un punto e l'altro
- la dimensione delle tavolette
- i numeri della tavoletta iniziale (vertice in basso a sinistra) e di quella finale (vertice in alto a destra) del bounding box.

Infine, viene calcolato il vertice max(x,y) del bounding box semplicemente sommando alle coordinate x,y di `min_xy` rispettivamente il numero di colonne e il numero di righe del bounding box.

La funzione `read_pts()`, dato in input un file `.PTS`, memorizza tutti i punti espressi in coordinate geografiche in un array, pronti per essere processati. I blocchi a diversa risoluzione vengono generati a partire da questi punti.

```
typedef struct F4_ {
    float x,y,z,w;
} F4;

typedef struct global_ {
    vector<F4> punti_m;
} global;

global g;
```

I punti vengono salvati nel vettore `g.punti_m`. In ogni elemento del vettore vegono salvati nei campi `x` ed `y` le coordinate cartesiane del punto, in `z` il livello di risoluzione a cui rappresentarlo e in `w` il tipo di condizione di bordo.

Anche in `multires` è presente il metodo `real_bln()` che memorizza in `pol.points` la lista dei vertici del poligono. Questo dipende dal fatto che in seguito tutti i punti del bounding box verranno divisi tra interni ed esterni al poligono e per effettuare questo controllo sarà necessario avere a disposizione i vertici.
Va precisato che in `bln_raster.PTS` sono presenti i vertici del poligono e che, calcolando il numero di punti fra un vertice e l'altro, potrebbero essere estratti senza consultare nuovamente il file `.BLN`.
Per come vengono interpolati i segmenti del poligono, però, il numero di punti fra due vertici non è costante, quindi risulta più semplice leggere il file `.BLN`. In uno scenario ideale, conoscendo il numero di punti fra due vertici consecutivi sarebbe possibile estrarli senza l'uso del `.BLN` avendo un aumento in efficienza ma anche in complessità delle operazioni.
Nell'economia dell'intero programma, ovvero di spazio in memoria da allocare, tuttavia questa operazione, che è un'operazione di input, non incide in maniera rilevante quindi è stata preferita la semplicità della lettura del file relativo al poligono.

3.2.2 Matrice a multi risoluzione

Il corpo principale del programma viene eseguito dalla funzione `multires()`. Per semplicità viene descritto per fasi successive, dato che si possono riconoscere diversi macro-passaggi che portano alla generazione della matrice a multi risoluzione.

Bitmask e seed points

Per prima cosa viene adattata la dimensione del bounding box così da poter suddividere la matrice in modo regolare, ovvero viene ricalcolato il punto max(x,y) in modo tale che la dimensione della matrice sia un multiplo della dimensione *BS* di un blocco.
Ogni livello di risoluzione viene rappresentato da una matrice di dimensione:

$$bsx_i = n/(BS * \Delta_i)$$
$$bsy_i = m/(BS * \Delta_i)$$

Queste matrici prendono il nome di bitmask dei livelli di risoluzione e con 4 livelli di risoluzione, le matrici vengono memorizzate in un unico array di 4 elementi nominato appunto `bitmask`.
Al livello di risoluzione 0, corrispondente alla massima risoluzione, la rispettiva matrice ha dimensioni

$$bsx * bsy$$

mentre al livello di risoluzione 3, cioè la risoluzione più bassa, ha dimensioni:

$$bsx/2^3 * bsy/2^3$$

Ogni elemento di una bitmask rappresenta nel bounding box un blocco di punti. Assumendo *BS = 16*, al livello 0, ad esempio, un elemento della matrice `bitmask[0][j]` rappresenta il rispettivo blocco di 16 punti in coordinate reali nel bounding box.
Un quadrato di 4 elementi della bitmask a livello i equivale ad un elemento della bitmask a livello di risoluzione i+1 (più bassa) e ad un quadrato di 8 nella matrice del livello precedente i-1 (più alta).

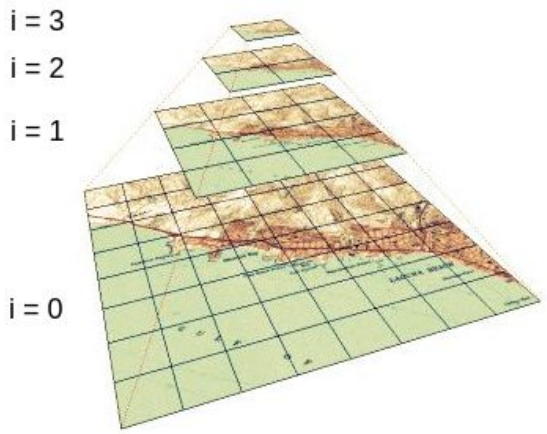


Figura 3.4: Esempio di bitmask dei diversi livelli

Per ogni punto del bounding box e per ogni risoluzione si possono ottenere le coordinate i,j del rispettivo blocco attraverso le formule

$$x_{blocco} = (x_p - x_{min})/(BS * dx)$$

$$y_{blocco} = (y_p - y_{min})/(BS * dy)$$

dove dx, dy rappresentano le dimensioni della cella nel bounding box (solitamente 1m x 1m) e x_{min}, y_{min} le coordinate di `min_xy`.
Una volta allocato l'array e inizializzati tutti gli elementi delle 4 bitmask, scorrendo il vettore `g.punti_m` vengono generati i **seed points**: per ogni livello di risoluzione si scorre tutto l'array dei punti, nei quali il livello di risoluzione a cui devono essere forzati è indicato nel campo z, e quando si lavora al livello corretto vengono calcolate le coordinate x_{blocco}, y_{blocco} dell'elemento della rispettiva matrice. Dato che ogni elemento del vettore `bitmask` logicamente è una matrice ma viene allocato come array monodimensionale, le coordinate ottenute vengono utilizzate per calcolare l'indice j di `bitmask[i][j]` corretto e all'elemento `bitmask[i][j]`, se non ancora assegnato o assegnato ad un livello maggiore (risoluzione più bassa!) viene assegnato il valore in z.
Quando viene assegnato un valore ad un elemento `bitmask[i][j]`, vengono assegnati allo stesso livello di risoluzione, se possibile, anche gli altri $2^i * 2^i - 1$ elementi costituenti un quadrato alla stessa risoluzione. Il concetto è quello di estendere la risoluzione del punto in maniera proporzionata al livello stesso, ovvero estendere per un'area piccola la risoluzione massima e per un'area grande la risoluzione più bassa. Quindi, per esempio prendendo *BS=16* la risoluzione di un punto forzato alla massima risoluzione possibile viene estesa a soli altri 3 blocchi, ovvero in totale 16*4

= 64 punti del bounding box, e quella di un punto forzato alla risoluzione più bassa viene estesa ad un'area di 8x8 blocchi, ovvero (16*8)x(16*8) = 16384 punti del bounding box.

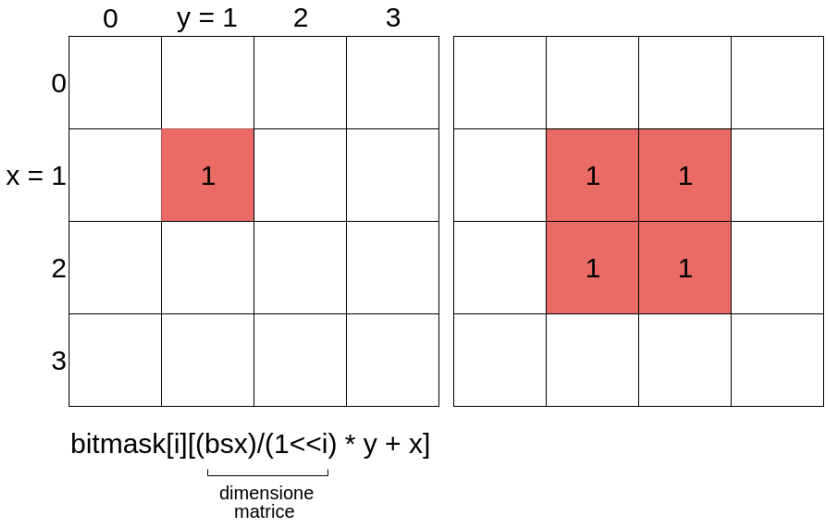


Figura 3.5: Esempio di completamento quadrato in bitmask (livello 0)

Completamento di bitmask a partire da seed points

Quelli che vengono chiamati seed points sono, quindi, in realtà i blocchi di punti in cui sono contenuti.
Quando la prima fase termina, il risultato ottenuto sono 4 matrici di zeri eccezion fatta per la matrice per i quadrati generati dai seed points. Le bitmask vengono completate in questa successiva fase nella quale, inizialmente, partendo dalla risoluzione più alta ogni quadrato generato viene copiato negli altri livelli. Un quadrato copiato a livello successivo viene dimezzato, mentre quando viene copiato in un livello precedente viene completato per raddoppiarne le dimensioni.
Quando questo lavoro viene terminato, l'utlimo step è quello di sostituire gli zero rimannenti con il valore più alto di `bitmask`, ovvero con il valore di risoluzione più bassa.
Il completamento delle bitmask avviene quindi lavorando da risoluzione più alta a risoluzione più bassa, *"provando"* a diminuirla per gli elementi per cui non ci sono motivi di mantenerla alta.
Viene inoltre segnalato errore se sono presenti conflitti in cui la richiesta di risoluzione per un elemento di una matrice è diversa per più di un livello rispetto a quella prevista.

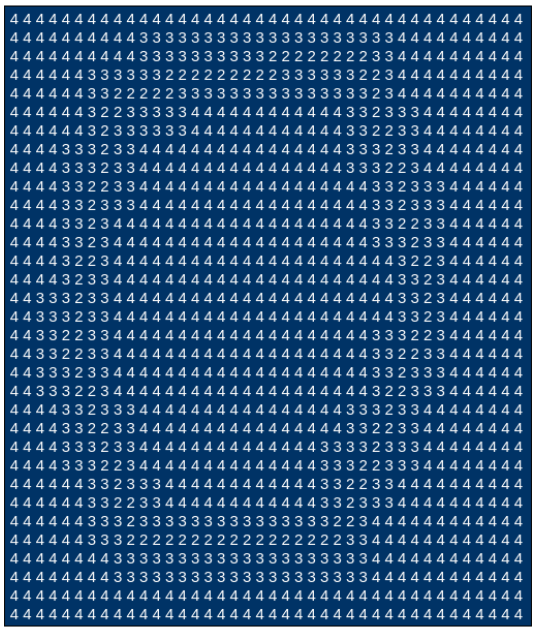


Figura 3.6: Esempio di bitmask completa del livello 2 di risoluzione

Codifica dei blocchi

Assieme all'array `bitmask` viene allocato un vettore del tutto analogo chiamato `bitmaskC` in cui, però, vengono memorizzate informazioni diverse.

Quando le bitmask sono complete, i blocchi su cui si lavora nel seguito di `mutires()` sono quelli che hanno valore uguale al livello di risoluzione e per identificarli in modo univoco vengono codificati. Ogni elemento `bitmaskC[i][j]` contiene un valore che può essere uguale ad un numero *x* o a *-1* a seconda che il rispettivo blocco `bitmask[i][j]` abbia valore uguale al livello di risoluzione i o meno (i+1 in realtà, dato che i va da 0 a 3 mentre i valori dei blocchi vanno da 1 a 4).
I blocchi vengono numerati partendo dal livello di risoluzione più alto fino a quello a risoluzione più basso dunque, in genere, più è piccolo il numero più è alta la risoluzione. Nell'esempio riportato in figura 3.6 i tre puntini vengono stampati al posto dei valori di `bitmaskC[i][j]` uguali a -1. Osservando bene bitmask, si nota come i blocchi relativi al perimetro del poligono siano stati forzati ad una risoluzione più alta, mentre guardando `bitmaskC` si può vedere come emerga nitidamente la figura del poligono. A questo livello di risoluzione, ovvero a risoluzione più bassa, ogni blocco rappresenta 256 punti in coordinate reali!

Quando vengono numerati i blocchi, viene anche calcolato il totale dei

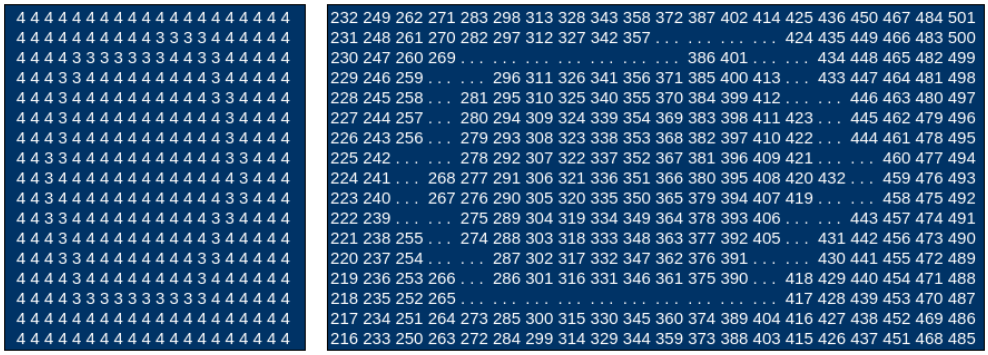


Figura 3.7: Esempio di bitmask e bitmaskC con i = 3

blocchi codificati e salvato nella variabile `tot_blocks`.
Il numero totale dei blocchi codificati viene utilizzato per calcolare il numero di righe `x_blocks` e il numero di colonne `y_blocks` della matrice a multi risoluzione. Inoltre, `x_blocks` viene calcolato così da essere una potenza di 2 e, di conseguenza, il prodotto fra numero di righe e numero di colonne è il più piccolo multiplo della potenza di 2 calcolata `x_blocks` maggiore di `tot_blocks`.

Un primo confronto

Mettendo a confronto questa nuova soluzione con quella già esistente la differenza evidente sta nella notevole riduzione della matrice a multi risoluzione. In questo lavoro di tesi, fino a questo momento infatti si è operato sul boinding box mentre nella versione originale si lavorava sull'intera mappa e fin dalle prime stampe che si possono eseguire su questa frazione di codice si può notare che, confrontando le dimensioni delle due matrici a multi risoluzione, la "nuova" ha dimensioni molto inferiori.
Nel codice di partenza i seed points dai quali vengono completate le bitmask sono indicati dall'utente tramite un file `.PTS`, mentre in questa nuova versione viene interpolato il perimetro del poligono ricavando i punti in modo automatico. Salvandoli in un file .PTS dal nome costante e usandoli come seed points vengono create le bitmask dei 4 livelli di risoluzione senza intervento da parte dell'utente, diminuendo la possibilità di commettere errore da parte di quest'ultimo. Devo aggiungere qualcos'altro?

Divisione tra punti interni ed esterni

La matrice dei blocchi in cui questi vengono memorizzati in ordine di codice si chiama `map.host_info`.
Le celle delle varie bitmask rappresentano blocchi (di punti) di dimensione diversa. Per uniformare tale rappresentazione all'interno della matrice `map.host_info`, tutti i blocchi vengono memorizzati con dimensione *BS*x*BS*. Le celle dei blocchi ai diversi livelli di risoluzione i rappresentano dunque i punti a distanza Δ_i (vedi definizione di blocco) del bounding box.

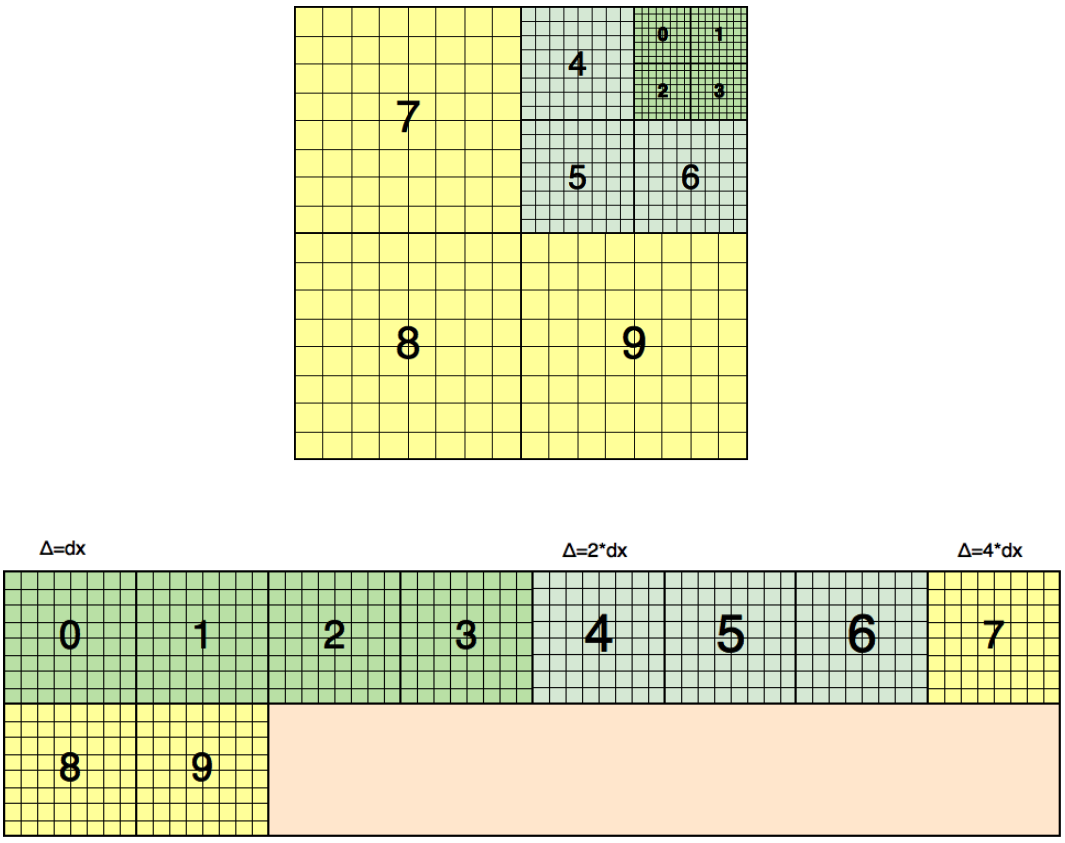


Figura 3.8: Esempio di map.host_info

Come le bitmask, anche `map.host_info` viene allocata come array monodimensionale mentre viene trattata logicamente come matrice. Per ogni punto in coordinata reale è possibile passare, come già mostrato, a coordinate di blocco della relativa bitmask. Attraverso queste coordinate è possibile conoscere il codice del blocco corrispondente di `bitmaskC[i][j]` e per mezzo del codice si possono calcolare gli indici i_{map}, j_{map} del blocco

all'interno della matrice `map.host_info` utilizzando le formule:

$$i_{map} = codice\%x\_blocks$$

$$j_{map} = codice / x\_blocks$$

Scorrendo ogni blocco come una matrice con indici x_b, y_b che vanno da 0 a BS (ovvero gli offset del blocco) si può ottenere l'indice `idx` nell'array monodimensionale `map.host_info` con la seguente formula:

$$idx = (j_{map} * BS + y_b) * BS * x\_blocks + (BS * i_{map} + x_b)$$

Dato che l'obiettivo è quello di lavorare solo nell'area interna al poligono, i punti dei blocchi della matrice `map.host_info` contengono un valore che può essere uguale a 0, cioè punto interno, oppure uguale a un valore `BIT_EXTERN`, che indica che il punto è esterno al poligono. Vengono, quindi, in primo luogo inizializzati ad un valore di default `ZERO` tutti gli elementi di `map.host_info`. Si potrebbe ora scorrere l'intera matrice per controllare ogni punto ed assegnarvi il valore 0 o il valore 1 a seconda che questo stia dentro o fuori al poligono, ma avendo la matrice di blocchi dimensioni relativamente grandi, sarebbe necessario un significativo tempo di calcolo da parte della macchina per eseguire tale operazione.
Viene, perciò, utilizzata una procedura ottimizzata in cui viene effettuata una prima suddivisione fra punti interni ed esterni su un set di punti di partenza. È utile eseguire questa operazione sull'insieme dei punti di tutti i blocchi in cui sono contenuti gli elementi del vettore `g.punti_m`, ovvero i blocchi che contengono l'interpolazione del perimetro del poligono. Questi blocchi prendono il nome di `bounding_blocks`.

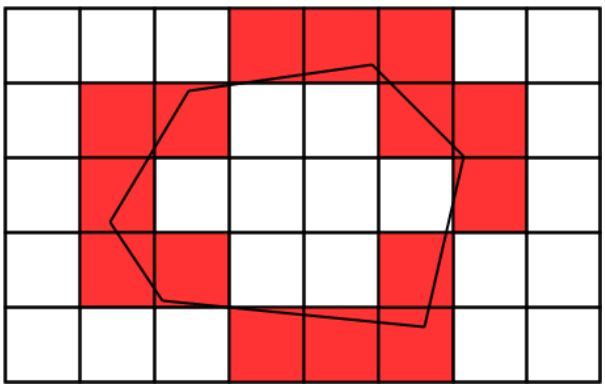


Figura 3.9: Esempio di bounding blocks

Vengono usati dunque due vettori di supporto di uguale lunghezza chiamati:

```
typedef struct int4_ {
    int x,y,z,w;
} int4;

vector<point> r_pt_list;
vector<int4> pt_list_info;
```

in cui per ogni punto del vettore `g.punti_m` si ricava il codice di blocco e scorrendo il blocco coi due indici x_b, y_b che vanno da 0 a *BS-1*, viene salvato in:

- pt_list_info[i].x, la coordinata x_b all'interno del blocco corrispondente
- pt_list_info[i].y, la coordinata y_b
- pt_list_info[i].z, il codice del blocco
- pt_list_info[i].w, non viene al momento utilizzato
- r_pt_list[i].x, la coordinata geografica x del punto rappresentata dalla cella del blocco
- r_pt_list[i].y, la coordinata geografica y.

Le informazioni dei punti di ciascun `bounding_block` vengono inserite una e una sola volta all'interno dei due array. Questo è possibile grazie all'utilizzo di un vettore di supporto "`assegnato`" di lunghezza `tot_blocks` in cui ogni elemento `assegnato[i]` ha valore 1 o 0 a seconda che il blocco con `codice = i` sia già stato processato o meno. Per ogni punto viene quindi ricavato il codice del blocco a cui appartiene e solo se `assegnato[codice] = 0` viene processato il blocco.

Viene valutata la posizione degli elementi in `r_pt_list`, rappresentanti i punti in coordinate reali, rispetto al poligono, ovvero, scorrendo il vettore `pol.points`, per ogni coppia di veritici viene analizzato se il punto è interno o esterno. Per ogni punto in coordinate reali, attraverso le informazioni degli elementi in `pt_list_info` è possibile calcolare l'indice `idx` dell'elemento corrispondente in `map.host_info` e, a seconda della posizione del punto reale rispetto al poligono, all'elemento in posizione `idx` viene assegnato il valore temporaneo `IN` o `OUT`.
Le informazioni utili a costruire l'indice `idx` vengono inserite in una delle due code:

```
vector<int4> queue;      // coda dei punti esterni al poligono
vector<int4> queue0;     // coda dei punti interni
```

Eseguendo queste operazioni per tutti gli elementi di `pt_list_info` e `r_pt_list` (scorrendoli simultaneamente), vengono riempite le code.
Con questo procedimento vengono concettualemente inseriti in `queue0` i punti che rappresentano il perimetro interno del poligono e in `queue` quelli rappresentanti il perimetro esterno.

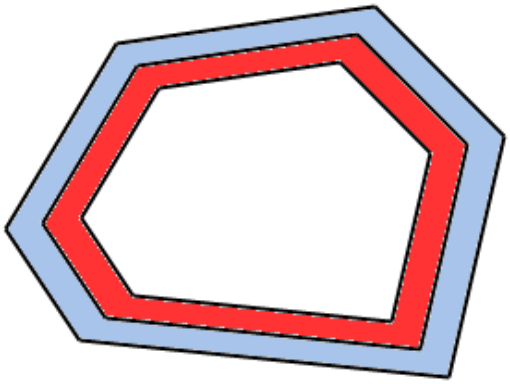
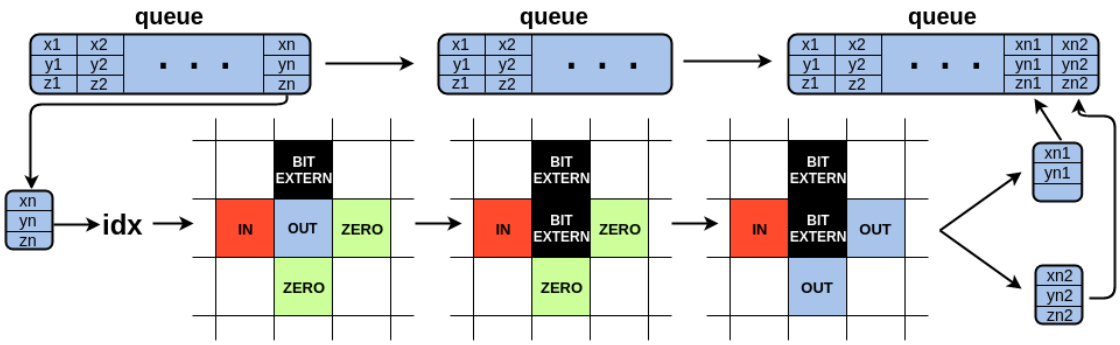


Figura 3.10: I punti dei `bounding_blocks` compongono il doppio perimetro del poligono

La coda `queue` contiene, come detto, tutti i punti dei `bounging_blocks` esterni al poligono. Estraendo elementi dalla coda fino a quando non è vuota, per ogni elemento estratto viene ricostruito l'indice `idx` e viene assegnato a `map.host_info[idx].x` il valore `BIT_EXTERN`. Per ogni punto `map.host_info[idx]` vengono controllati i vicini Nord/Sud/Est/Ovest e per ognuno dei vicini, se il valore è uguale a `ZERO`, viene assegnato il valore `OUT` e vengono inserite le informazioni del punto in coda a `queue`.



Piuttosto che scorrere l'intera matrice cercando i punti interni ed esterni, con questo tipo di procedimento da un set di punti di partenza viene espansa la ricerca ai vicini. Inoltre, per ogni coda, la ricerca viene condotta solo nei sensi corretti, ovvero, prendendo l'esempio della figura 3.11,

dal perimetro esterno verso i lati di `map.host_info` e dal perimetro interno verso il centro della matrice.

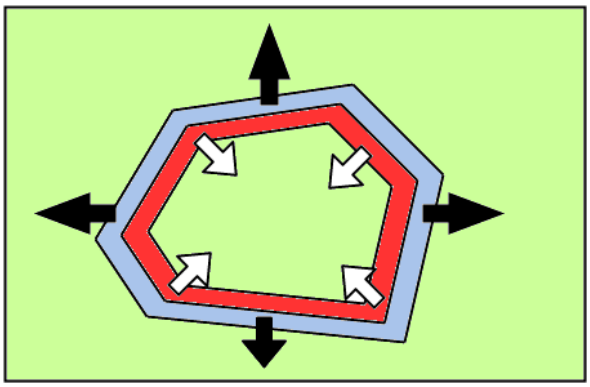


Figura 3.11: Espansione della ricerca

Per esplorare i vicini viene utilizzato un array di supporto chiamato `map.neigh` lungo `4*tot_blocks`:

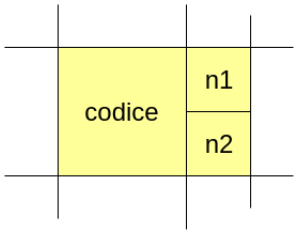
```
typedef struct {
    char lev;
    int n1;
    int n2;
} neigh_t;

neigh_t* map.neigh;
```

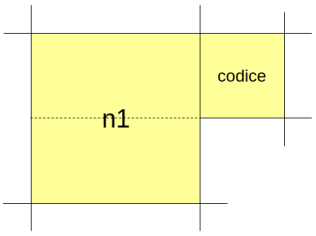
per ogni blocco, identificato dal proprio codice, vengono salvati negli elementi `map.neigh[4*codice+ii]` le informazioni relative ai propri vicini N/S/W/E, ovvero:

- in `map.neigh[i].lev` il livello di risoluzione del vicino (0 stessa risoluzione, 1 risoluziione minore, -1 risoluzione maggiore)
- in `map.neigh[i].n1` il codice di blocco del vicino
- in `map.neigh[i].n2` l'eventuale codice di blocco del secondo vicino (-1 di default)

Se il vicino di blocco è alla stessa risoluzione in `n1` viene inserito il codice del vicino ed `n2` viene lasciato uguale a -1.
Nel caso in cui un blocco abbia come vicino uno a risoluzione maggiore, allora il numero dei vicini è uguale a 2 e dunque, solo in questo caso, al



campo `n2` viene assegnato il codice di blocco del secondo vicino.
Se, al contrario, il vicino ha risoluzione minore, significa che rappresenta un blocco di dimensioni doppie e il campo `n2` viene usato per indicare la corretta metà di adiacenza del blocco, ovvero 0 per la prima metà e 1 per la seconda.



Con le informazioni degli elementi del vettore `map.neigh` vengono ricostruiti gli indici `idx1` ed eventualemente `idx2` dei vicini di ogni blocco e viene eseguita l'espansione della ricerca.

Il processo è speculare per la coda dei punti interni, in cui l'espansione viene eseguita esclusivamente per i punti interni al poligono.

Condizioni di bordo

Quando le code sono state svuotate tutti i punti in `map.host_info` hanno valore `BIT_EXTERN` o 0.
Le celle sul bordo del poligono devono contenere le condizioni di bordo ed oltre a queste, è necessario sapere a quali posizioni dei vicini questi valori fanno riferimento.
Ogni cella può avere 1, 2 o 3 vicini esterni. Dato che i punti con un singolo vicino o quelli con tre vicini esterni non sono frequenti, vengono calcolate le condizioni di bordo delle sole celle con due vicini esterni.
Per ogni punto di `map.host_info` viene controllato che sia interno e se stia sul bordo e, contando quanti vicini hanno valore `BIT_EXTERN`, solo a quelli che hanno due vicini esterni viene assegnato il valore temporaneo 2.
Dopo una prima spazzata sulla matrice di blocchi, viene eseguita una seconda lettura dei valori delle celle e ad ogni cella con valore uguale a 2,

viene assegnato un numero che rappresenta una combinazione dei valori `BIT_N`, `BIT_S`, `BIT_W`, `BIT_E` a seconda di quali siano i vicini coinvolti. Nel campo y dell'elemento generico `map.host_info[idx]` viene salvato (non so cosa), mentre nei campi z e w vengono salvati i tipi di condizione al contorno (vedi "Poligono" nella parte introduttiva) relativi ai vicini coinvolti.

Caricamento delle tavolette