Indice

Indice degli algoritmi

Capitolo I

Geometria Computazionale

La *geometria computazionale* affronta una serie di problemi molto diversi (ad esempio la ricerca della coppia di punti più vicini in un insieme di punti, la ricerca dell'intersezione di segmenti, la costruzione dell'inviluppo convesso di un insieme di punti sparsi nel piano o nello spazio), che possono avere applicazioni pratiche molto diversificate.

Possibili applicazioni possono ad esempio riguardare:

- **Problemi di motion planning e visione** Si applicano ad esempio in robotica, quando si vuole pianificare il percorso di un robot tra ostacoli o calcolare l'ingombro per determinare (o prevenire) il contatto con ostacoli.

- **Sistemi GIS** Si utilizza per esempio nella sovrapposizione di mappe, per determinare punti particolari, come l'intersezione di strade.

- **Computer graphics** Viene utilizzata nella realizzazione di applicazioni e ambienti 3D o di realtà virtuale, per generare ombre, luci o calcolare il movimento all'interno dell'ambiente.

- **Statistica e demografia** Un esempio è la dislocazione dei distretti elettorali, da distribuire in base all'addensamento della popolazione in certe aree.

- **Biologia computazionale** Un possibile utilizzo è la rappresentazione 3D di molecole o della catena del DNA.

I.1 Caratteristiche degli algoritmi

Gli algoritmi di geometria computazionale presentano le seguenti peculiarità:

- **Robustezza** Gli algoritmi devono essere robusti affinchè sia possibile utilizzarli nella pratica, ossia a fronte di una imprecisione di calcolo (ad esempio una misura errata dovuta a overflow della macchina) non deve produrre un output insensato.

- **Gestione implicita dei casi particolari** I casi particolari sono molto frequenti in geometria computazionale, e una loro gestione esplicita renderebbe il codice eccessivamente complesso e lungo da implementare.
- **Algoritmi output-sensitive** Per questi tipi di algoritmi non conta la complessità in sè, ma quella correlata alle dimensioni dell'output. Pertanto ci si aspetta che esso sia veloce se l'output è piccolo (ad esempio se l'insieme dei punti è piccolo), più lento se l'output è di grandi dimensioni.
- **Uso di strutture dati definite** Ogni algoritmo necessita (e fa uso) di strutture dati particolari e predefinite.

I.2 Elementi base

Gli elementi base che si utilizzano in geometria computazionale sono:

- **Punto** Viene definito dalle sue coordinate cartesiane. Ad esempio: $p_1 = (x_1, y_1)$, $p_2 = (x_2, y_2)$, $p_3 = (x_3, y_3)$.
- **Segmento** È definito come l'insieme di tutti i punti che possono essere descritti come combinazione convessa dei punti base p_1 e p_2. $\overline{p_1p_2} = \{(x, y) : x = \lambda x_1 + (1 - \lambda)x_2, y = \lambda y_1 + (1 - \lambda)y_2, 0 \leq \lambda \leq 1\}$
- **Segmento orientato** È un segmento dotato di verso. Si indica con $\overrightarrow{p_1p_2}$.

I.3 Domande di base

In geometria computazionale si possono formulare tre domande fondamentali, alle quali si può dare risposta ricorrendo a procedure che verranno utilizzate in diversi contesti.

- **Domanda 1** Dati due segmenti orientati con un vertice in comune, $\overrightarrow{p_0p_1}$ e $\overrightarrow{p_0p_2}$: *quando è possibile dire che un segmento si trova a destra dell'altro?* **Nota:** un segmento è a destra rispetto ad un altro quando è immediatamente successivo nella rotazione oraria (nella figura I.1 $\overrightarrow{p_0p_2}$ è a destra di $\overrightarrow{p_0p_1}$).

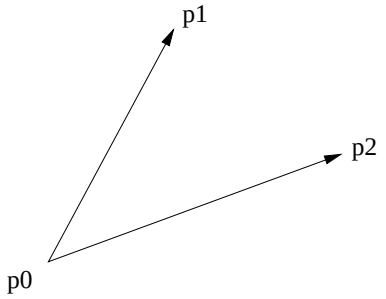

Figura I.1. Esempio di posizionamento di segmenti

- **Domanda 2** Dati 3 punti p_0, p_1 e p_2 e i segmenti $\overline{p_0p_1}$ e $\overline{p_1p_2}$:
  quando, percorrendo i segmenti in successione da p_0 a p_2 viene effettuata una voltata a destra (o a sinistra)?

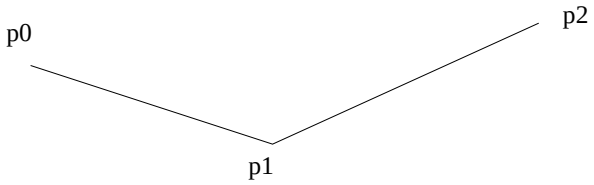

Figura I.2. Esempio di voltata a sinistra percorrendo i segmenti in successione partendo da p_0

- **Domanda 3** Dati due segmenti distinti $\overline{p_1p_2}$ e $\overline{p_3p_4}$:
  quando i due segmenti si incrociano?

Ovviamente si vuole rispondere a queste domande in tempo possibilmente costante, e in modo da evitare errori.

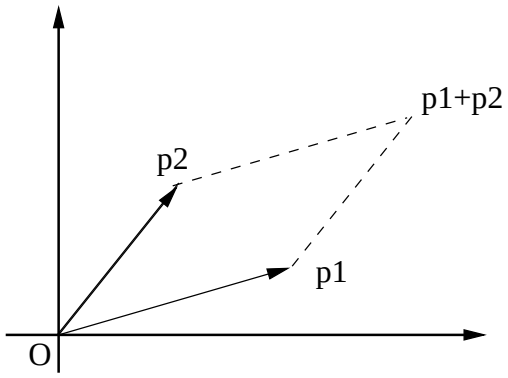

Figura I.3. Parallelogramma la cui area corrisponde al modulo del prodotto vettoriale

Per rispondere a queste domande si usa la seguente proprietà. Considerati i vettori $\overrightarrow{p_1}$ e $\overrightarrow{p_2}$ uscenti dall'origine, il loro prodotto $\overrightarrow{p_1} \times \overrightarrow{p_2} = det \begin{bmatrix} x_1 & x_2 \\ y_1 & y_2 \end{bmatrix} = x_1y_2 - x_2y_1$ corrisponde all'area del parallelogramma definito in figura I.3, e il segno di questo prodotto corrisponde alla seguente casistica:

se $\overrightarrow{p_1} \times \overrightarrow{p_2} > 0$ allora p_1 è a destra di p_2;
se $\overrightarrow{p_1} \times \overrightarrow{p_2} < 0$ allora p_1 è a sinistra di p_2;
se $\overrightarrow{p_1} \times \overrightarrow{p_2} = 0$ allora p_1 e p_2 stanno sulla stessa retta che passa per l'origine, ossia sono collineari con O.

A questo punto si può rispondere alle precedenti domande in questo modo:

- **Domanda 1** Collocata l'origine nel punto p_0, si considera il prodotto $(p_2 - p_0) \times (p_1 - p_0)$ e si valuta il segno.

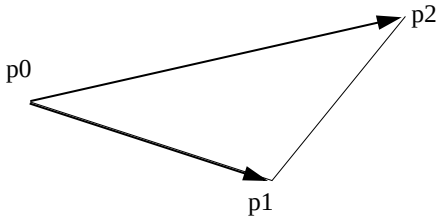

Figura I.4. Schema per stabilire se percorrendo i segmenti da p_0 a p_2 si effettua una voltata a destra (nel caso della figura si effettua una voltata a sinistra)

- **Domanda 2** Si considerano i vettori $\overrightarrow{p_0p_1}$ e $\overrightarrow{p_0p_2}$, riconducendosi quindi alla domanda 1.

- **Domanda 3** Consideriamo le rette che contengono i segmenti $\overline{p_1p_2}$ e $\overline{p_3p_4}$. I due segmenti si intersecano se p_1 sta in un semipiano delimitato dalla retta contenente $\overline{p_3p_4}$ e p_2 nell'altro. La stessa cosa vale per p_3 e p_4 nei confronti della retta contenente $\overline{p_1p_2}$. Si utilizza quindi il seguente algoritmo:

Algoritmo I.1 SegmIntersection(P)

1: $d_1 \leftarrow (p_1 - p_3) \times (p_4 - p_3)$
2: $d_2 \leftarrow (p_2 - p_3) \times (p_4 - p_3)$
3: $d_3 \leftarrow (p_3 - p_1) \times (p_2 - p_1)$
4: $d_4 \leftarrow (p_4 - p_1) \times (p_2 - p_1)$
5: **if** (($d_1 > 0$ and $d_2 < 0$) or ($d_1 < 0$ and $d_2 > 0$)) and (($d_3 > 0$ and $d_4 < 0$) or ($d_3 < 0$ and$d_4 > 0$)) **then**
6: TRUE
7: **else**
8: *... bisogna considerare tutti i casi particolari in cui una delle d_i si azzera ovvero quando uno dei punti giace sulla retta*

I.4 Calcolo dell'inviluppo convesso di un insieme di punti

Dato un insieme di punti $P = \{p_1, \ldots, p_n\}$, si chiama poliedro convesso (convex hull), e lo si indica con $CH(P)$, il poliedro convesso più piccolo che racchiude tutti i punti di P.

Esistono una grande quantità di algoritmi che calcolano l'inviluppo convesso di un insieme di punti, che si distinguono per correttezza, efficienza e robustezza. Nel seguito vedremo i seguenti algoritmi:

- Algoritmo Slow Convex Hull
- Algoritmo Graham Scan
- Algoritmo Gift Wrapping o di Jarvis-March

I.4.1 Algoritmo Slow Convex Hull

Prima di entrare nel dettaglio degli algoritmi, vediamo come rappresentare l'inviluppo convesso $CH(P)$. Infatti la rappresentazione individuata dalla definizione data prima non è di utilità pratica. Un modo di rappresentare $CH(P)$ potrebbe essere dato dal sottoinsieme $P' \subseteq P$ dei punti esterni di $CH(P)$. Ancora più utile potrebbe essere ordinare P' in base ad un ordine orario (o antiorario). L'osservazione fondamentale che consegue da questo tipo di rappresentazione è che dati due punti consecutivi p_1 e p_2 di $CH(P)$, ogni altro punto $p \in P$ è a destra del segmento $\overrightarrow{p_1p_2}$.

I dati utilizzati dall'algoritmo sono i segmenti via via trovati che appartengono al bordo. Essi vengono memorizzati un una lista, che chiamiamo E, e che al termine viene ordinata in senso orario.

Algoritmo I.2 SlowConvexHull(P)

```
1: E ← ∅
2: for each p, q ∈ P, p ≠ q do
3:    valid ← TRUE
4:    for r ∈ P, r ≠ q and r ≠ p do
5:       if r è a sinistra di pq→ then
6:          valid ← FALSE
7:          break
8:       else if valid then
9:          E ← E ∪ {pq→}
10: L ← sort(E)
11: RETURN(L)
```

Analisi di complessità I cicli for annidati, ciascuno fatto n volte, hanno globalmente complessità $O(n^3)$. L'ordinamento di L ha complessità $O(h \lg h)$, dove $|E|$ rappresenta la dimensione dell'output. SlowConvexHull ha quindi una complessità dell'ordine di $O(n^3)$.

Caratteristiche

- Presenta complessità elevata.
- Non è output-sensitive.
- Non è robusto. Se si considerano tre punti collineari p_0, p_1 e p_2 l'algoritmo considera i segmenti $\overline{p_0p_1}$, $\overline{p_1p_2}$ e $\overline{P_0P_2}$. Se a causa di errori di precisione della misura si ottiene che P_0 è a sinistra di $\overline{p_1p_2}$, p_1 è a sinistra di $\overline{p_0p_2}$ e p_2 è a sinistra di $\overline{p_0p_1}$, i tre segmenti vengono scartati, e non c'è più modo di reinserirli. L'output finale risulta essere un poligono non chiuso!
- Bisogna trattare i casi particolari in maniera esplicita. Sempre considerando il caso dei tre punti collineari p_0, p_1 e p_2, bisogna in qualche modo eliminare il punto centrale p_1 ma l'algoritmo non è in grado di farlo in modo esplicito.

I.4.2 Algoritmo Graham Scan

Questo algoritmo è incrementale. Si parte da un punto che sicuramente appartiene al bordo (ad esempio da quello con coordinate x e y minime) e ad ogni iterazione si va a considerare un punto aggiuntivo; si valuta se questo punto sta a destra o a sinistra rispetto al segmento immediatamente precedente (in base alla misura dell'angolo tra i segmenti). Se non si effettua una voltata a sinistra, il punto appartiene al bordo, altrimenti il punto non appartiene al bordo e va a scartato.

La struttura dati più adatta per implementare questo algoritmo è la pila, in cui memorizzo i punti estremi candidati, infatti si va sempre a considerare l'ultimo elemento (o al massimo il penultimo).

L'algoritmo è il seguente:

Algoritmo I.3 GrahamScan(P)

```
1: p_0 ← punto con coordinata y minima
2: p_1 ... p_n ordinati in senso orario rispetto a P_0
3: Push(p_0, S)
4: Push(p_1, S)
5: Push(p_2, S)
6: for i ← 3 ... n do
7:    while angolo NextToTop(S), Top(S), p_i non gira a destra do
8:       Pop(S)
9:    Push(S, p_i)
10: RETURN (S)
```

La correttezza dell'algoritmo, vista la natura incrementale, si dimostra facilmente per induzione.

Analisi di complessità L'ordinamento di $p_1 \dots p_n$ avviene in $O(n \lg n)$. Si può affermare in base a tecniche di analisi di complessità ammortizzata, che il ciclo *while* ha un costo di $O(n)$ (infatti una volta fatta una Pop a un punto, questo non viene più introdotto). L'operazione dominante è l'ordinamento, pertanto la complessità dell'algoritmo è $O(n \lg n)$.

Caratteristiche

- Più robusto dell'algoritmo SlowConvexHull. Non è ancora esente da errori, ossia può introdurre ancora un inviluppo non convesso, ma comunque l'output è un poligono chiuso.
- Come si vede dalla complessità, è più efficiente, e viene eseguito in tempo quasi lineare.
- Non è output sensitive.
- Il caso di punti collineati è trattato in modo implicito.

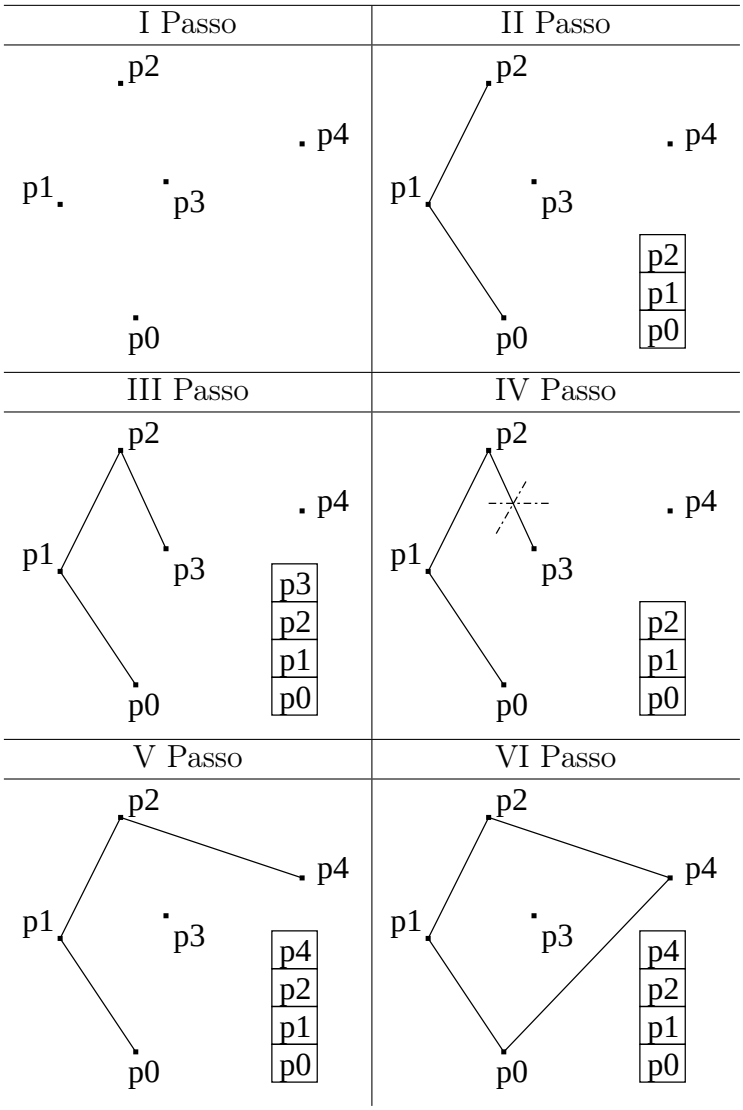

Tabella I.1. Esempio di applicazione dell'algoritmo GrahamScan

I.4.3 Algoritmo Gift-Wrapping o di Jarvis-March

Questo algoritmo usa un procedimento simile a quello che si usa nella vita reale quando si vuole incartare un pacco da regalo (da cui il nome). Esso inizia da un punto p_0 appartenente al bordo (ad esempio quello di coordinate minime) e trova il punto con il più piccolo angolo polare maggiore di zero rispetto a p_0. Questo continua fino a quando si raggiunge il punto p_h, apice dell'inviluppo. Dopodichè viene ricercato il più grande angolo polare con segno negativo. Una volta che si raggiunge nuovamente il punto p_0 l'algoritmo termina. L'algoritmo definisce due insiemi, nel seguito chiamati come $Gruppo1$ e $Gruppo2$, contenenti i punti rispettivamente da p_0 a p_h e da p_{h+1} a p_0.

Analisi di complessità La ricerca del punto minimo impiega $O(n)$. Il calcolo degli angoli, che si è visto essere il punto cruciale dell'algoritmo, viene eseguito anch'esso in

Algoritmo I.4 JarvisMarch(P)

```
1: p_0 ← punto con coordinate x e y minime
2: p_x ← p_0
3: Gruppo ← 1
4: while p_x ≠ p_0 do
5:    Trova l'angolo polare dei punti in P \ {p_x}
6:    if Gruppo = 1 then
7:       p_y ← punto con il minor angolo polare ≥ 0
8:       if ∄p_y then
9:          Gruppo ← 2
10:   if Gruppo = 2 then
11:      p_y ← punto con il maggior angolo polare ≤ 0
12:   S ← S + {p_y}
13:   p_x ← p_y
14: RETURN(S)
```

tempo costante, cioè in $O(n)$. Il procedimento va ripetuto per tutti gli h punti che costituiscono il bordo, quindi la complessità è $O(hn)$.

Caratteristiche

- Se h non è troppo elevato, l'algoritmo risulta abbastanza efficiente. Al contrario, se tutti i punti sono disposti sul bordo, $h \to n$ e il tempo di esecuzione diventa $O(n^2)$ per cui in questo caso è preferibile l'algoritmo GrahamScan. Se però si sa in anticipo che h è piccolo rispetto a n, allora l'algoritmo risulta efficiente.

- È un algoritmo output-sensitive.

I.5 Intersezione di segmenti

Questo problema consiste, dato un insieme di segmenti disposti nel piano (o nello spazio), di determinare i punti di intersezione. Questo problema ha molti ambiti applicativi, ad esempio nei sistemi GIS per determinare intersezioni tra strade o percorsi in una mappa, ed è alla base di problemi più complessi come la sovrapposizione di mappe.

Quello che si vuole trovare è un algoritmo efficiente che esegua il calcolo delle intersezioni. Dati n segmenti, al massimo si possono avere n^2 intersezioni. Un algoritmo semplice (ma inefficiente) può andare a considerare ogni coppia di segmenti e verificare se esiste un'intersezione tra di essi. Un algoritmo di questo tipo impiega un tempo di esecuzione $O(n^2)$: sebbene abbia complessità ottimale nel caso pessimo, non è output sensitive e, se le intersezioni sono poche, risulta poco efficiente.

I.5.1 Algoritmo della Sweep Line

Un algoritmo più efficiente consiste nell'inserire i segmenti in un sistema di riferimento cartesiano. Quindi si introduce una retta verticale, la *sweepline*, che si muove nella direzione dell'asse delle ascisse. Ogni volta che viene incontrato un punto estremo di un segmento o di intersezione viene generato un evento dell'algoritmo. L'input dell'algoritmo è rappresentato da tutti i punti estremi dei segmenti, l'output è dato dai punti di intersezione.

Nel seguito verranno introdotte alcune ipotesi semplificative.

- La sweep line è verticale. Questo implica che non devono esistere segmenti (e quindi intervalli) verticali. Se dovessero esserci, si pensa la sweep line ruotata di un ϵ in modo che i due estremi vengano comunque incontrati in istanti diversi.
- In ogni punto si intersecano al più due segmenti.

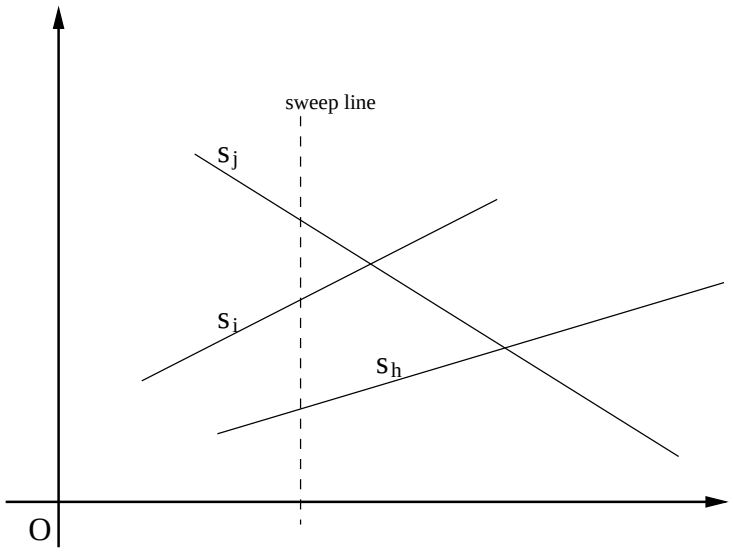

Figura I.5. Intersezione di segmenti e sweep line

L'idea dell'algoritmo si basa sulla seguente osservazione. Considerato un segmento s_i tagliato dalla sweep line ad una certa coordinata y_i possiamo limitarci a controllare le intersezioni di s_i con gli intervalli s_j e s_h rispettivamente immediatamente sopra e immediatamente sotto nella sweep line. Lo stato della sweep line non si modifica fino a quando non si verificano i seguenti eventi:

- un nuovo intervallo viene incontrato;
- un intervallo esce dalla sweep line;
- si verifica una intersezione.

In tutti e tre i casi, modificandosi lo stato della sweep line, dobbiamo aggiornare lo stato delle adiacenze tra segmenti e fare nuovi controlli sulle intersezioni. Si noti in particolare

come quando abbiamo un intersezione l'ordine dei due segmenti intersecantisi si inverte. L'algoritmo necessita di due strutture dati diverse:

- Una struttura dati dinamica (sweep line), che tiene conto dei segmenti incontrati che intersecano la sweep line in una data ascissa x, registrandone anche l'ordine di ordinata y.
- Una struttura dati (coda di eventi), che tiene conto degli eventi che generano modifiche alla struttura dati dinamica, quindi estremi sinistri e destri dei segmenti e punti d'intersezione.

Algoritmo I.5 FindIntersectionSweepLine(S)

```
1: T ← ∅ /*sweep line*/
2: Ordina S per x crescente e inseriscili in Q
3: for each p ∈ Q do
4:    if p è un estremo sinistro di un segmento s then
5:       Insert(T,s)
6:       if (∃ Above(T,s) and interseca s) or (∃ Below(T,s) and interseca s) then
7:          Calcola intesezione r
8:          Insert(Q,r)
9:    else if p è un estremo destro di un segmento s then
10:      Delete(T,s)
11:      if ∃ Above(T,s) and ∃ Below(T,s) and si intersecano then
12:         Calcola intesezione r
13:         Insert(Q,r)
14:   else if p è un'intersezione then
15:      Swap(intervalli che si intersecano in p)
```

Esempio di funzionamento Data la figura seguente, all'istante 1 viene trovato l'estremo destro di $s1$, quindi $s1$ viene inserito nella sweep line. All'istante 2 compare il segmento $s2$, quindi viene inserito nella sweep line e viene inserito nella coda il possibile evento di intersezione $I12$. Quindi in 3 compare il segmento $s3$, che separa $s1$ e $s2$. In 4 viene calcolata l'intersezione $I13$; in questo modo $s1$ e $s3$ si scambiano di posto nella sweep line, cosicchè $s1$ ed $s2$ si trovano nuovamente adiacenti nella lista, ed è possibile calcolare l'eventuale intersezione $I12$. Ma, siccome può esserci un solo evento per ogni intersezione, $I12$ non può essere inserito nuovamente nella coda. Così, quando un'intersezione deve essere inserita nella coda, bisogna prima effettuare una ricerca nella coda e verificare che non sia già stata inserita precedentemente. In questo modo nella coda esiste una sola intersezione per ogni coppia di segmenti, quindi l'etichettatura proposta è sufficiente ad identificare l'intersezione stessa.

Analisi di complessità Le operazioni di Insert, Delete e Riordina dei segmenti nella lista può essere fatta, adottando opportune strutture, in $O(\lg n)$. La dimensione

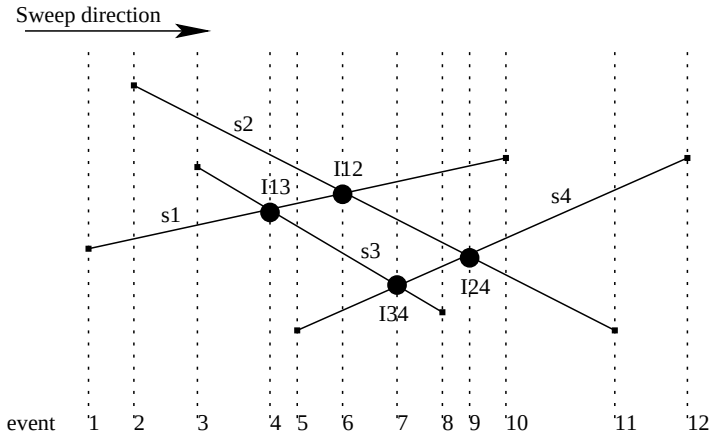

Figura I.6. Scansione di un insieme di segmenti attraverso l'algoritmo che utilizza una sweep line

massima della coda è, nel caso pessimo, $2n + k \leq 2n + n^2$, quindi anche l'inserzione nella coda può essere fatta in $O(\lg(2n + n^2)) = O(\lg n)$. Il punto critico sta nel come calcolare un'intersezione valida. L'assunzione che due segmenti si intersecano se vengono incontrati dalla sweep line nello stesso istante t non è sufficiente a rendere l'algoritmo efficiente. L'osservazione cruciale riguarda la posizione 'sopra-sotto' che due segmenti che si intersecano devono avere nella sweep line. In questo modo possono verificarsi i seguenti casi:

- Quando un segmento è inserito nella sweep line, bisogna verificare se si interseca con i suoi vicini sopra e sotto nella sweep line.
- Quando un segmento è cancellato dalla sweep line, i suoi vicini sopra e sotto della sweep line sono messi assieme come nuovi vicini, in modo da poter valutare la loro intersezione.
- All'evento di intersezione, lo scambio di posizioni nella sweep line permette di calcolare le intersezioni con i loro nuovi vicini.

Quindi per ogni evento (estremo o punto di intersezione) bisogna valutare al più due intersezioni. Se viene scelta come struttura dati quella dell'albero binario le operazioni possono essere fatte in $O(\lg n)$. Pertanto la complessit à dell'algoritmo diventa $O(ordinamentoiniziale) + O(processamento) = O(n \lg n) + O((2n + k) \lg n) = O((n + k) \lg n)$. Questo è vero se gli alberi sono bilanciati, altrimenti si rischia di peggiorare molto le prestazioni. Un metodo per garantire che gli alberi che vanno a rappresentare la struttura dinamica della sweep line siano e restino bilanciati è quello di costruire alberi colorati, come gli *alberi rosso-neri*.

I.5.2 Alberi binari e alberi binari colorati

Si presenta ora un breve ripasso sugli alberi binari e alberi binari colorati, per comprendere meglio come l'algoritmo della sweep line opera utilizzando alberi come strutture dati.

Un albero binario è costituito da nodi, ciascuno dei quali contiene un dato, o chiave (key) e tre puntatori: un puntatore $p(x)$, ossia al proprio genitore, un puntatore sinistro $left(x)$ e un puntatore destro $right(x)$ che puntano rispettivamente ai figli sinistro e destro del nodo. In un albero binario devono valere anche le seguenti regole:

$$key(x) \geq key(y) \text{ per ogni } y \in \text{albero in } left(x)$$
$$key(x) \leq key(y) \text{ per ogni } y \in \text{albero in } right(x)$$

ossia tutti i nodi appartenenti al sottoalbero sinistro di un nodo devono avere chiave minore o uguale a quella del nodo, quelli del sottoalbero destro chiave maggiore o uguale a quella del nodo.

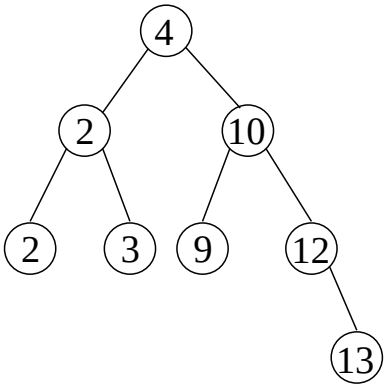

Figura I.7. Esempio di albero binario

Su un albero binario possono essere effettuate diverse operazioni. Le principali (e oltretutto necessarie per eseguire l'algoritmo della Sweep Line) sono le seguenti.

Visita L'algoritmo scorre tutti gli elementi in ordine non decrescente di chiave. È applicato in modo ricorsivo. Quello presentato è l'algoritmo di visita anticipata. Si possono definire anche altri tipi di visite. Applicato al grafo in figura I.7 restituisce la seguente lista:

2 2 3 4 9 10 12 13

Algoritmo I.6 VISITA(x)

```
1: if x ≠ NIL then
2:     VISITA(left(x))
3:     PRINT(key(x))
4:     VISITA(right(x))
```

Ricerca Serve a ricercare un particolare elemento (key) all'interno dell'albero. Essa impiega un tempo proporzionale all'altezza h dell'albero, cioè $O(h)$. Se l'albero è bilanciato $h = O(\lg n)$, se l'albero è un cammino $h = O(n)$.

Algoritmo I.7 TreeSearch(x, k)

```
1: if x = NIL or key(x) = k then
2:     RETURN(x)
3: else if k < key(x) then
4:     RETURN(TreeSearch(left(x)))
5: else
6:     RETURN(TreeSearch(right(x)))
```

TreeMin e TreeMax Queste due funzioni ritornano rispettivamente l'elemento minimo e massimo contenuti nell'albero.

Algoritmo I.8 TreeMin(x)

```
1: while left(x) ≠ NIL do
2:     x ← left(x)
3:     RETURN(x)
```

Algoritmo I.9 TreeMax(x)

```
1: while right(x) ≠ NIL do
2:     x ← right(x)
3:     RETURN(x)
```

TreeSucc e TreePrec Queste due funzioni servono a individuare l'elemento immediatamente successivo o precedente a x.

Insert Questa operazione serve ad inserire un nuovo elemento nell'albero. L'algoritmo crea un nuovo nodo con puntatori $left(x)$ e $right(x)$ a NIL. Eseguendo la ricerca trova la posizione in cui inserire il nodo. Il nuovo nodo viene sempre inserito come una nuova foglia. Nell'albero di esempio, se vogliamo inserire il nodo 11, si ottiene il risultato in figura I.8.

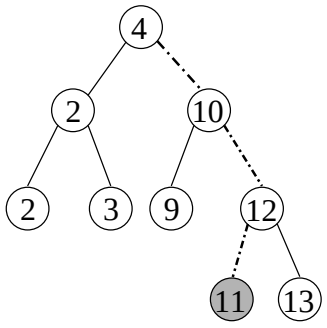

Figura I.8. Inserimento di un nodo con chiave 11 nell'albero della figura I.7

Algoritmo I.10 TreeSucc(x)

```
1: if right(x) ≠ NIL then
2:     RETURN(TreeMin(right(x)))
3: y ← p(x)
4: while y ≠ NIL and x = right(y) do
5:     x ← y
6:     y ← p(y)
```

Delete Questa funzione serve ad eliminare un nodo dall'albero. Questa operazione è più complessa, in quanto bisogna sempre garantire che, una volta eliminato un nodo, la struttura e l'ordinamento siano coerenti con la definizione di albero binario. Si possono presentare diversi casi.

Caso 1 : il nodo da cancellare non ha figli. In questo caso è sufficiente rimuoverlo. Se dal grafo ottenuto in figura I.8 vogliamo rimuovere il nodo 11, otteniamo il grafo di partenza.

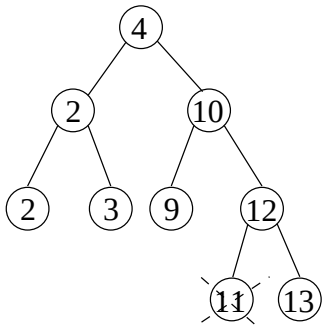

Figura I.9. Rimozione del nodo di chiave 11 dall'albero della figura I.8

Caso 2 : il nodo ha un solo figlio. In questo caso il figlio del nodo eliminato diviene figlio del padre del nodo eliminato. Se nel grafo dell'esempio vogliamo rimuovere il nodo 12 si ottiene quanto riportato in figura I.10.

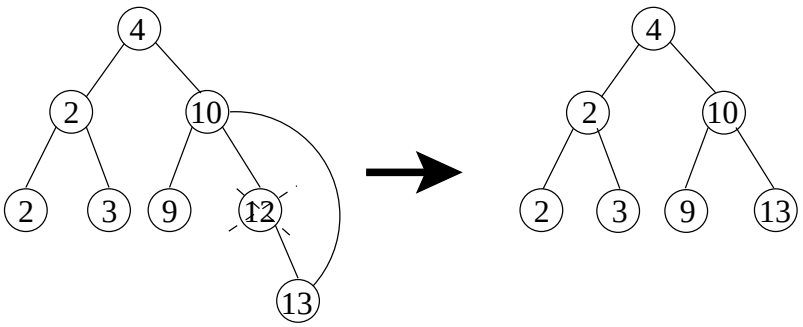

Figura I.10. Rimozione del nodo di chiave 12 dall'albero dell'esempio

Caso 3 : se il nodo da eliminare possiede due figli, si prende il primo discendente senza figli sinistri e lo si sostituisce al nodo eliminato. Ad esempio, volendo eliminare il nodo 10 spostiamo il successore senza figli sinistri di 10 (il nodo 12) nella posizione occupata da 10, ottenendo quanto riportato in figura I.11.

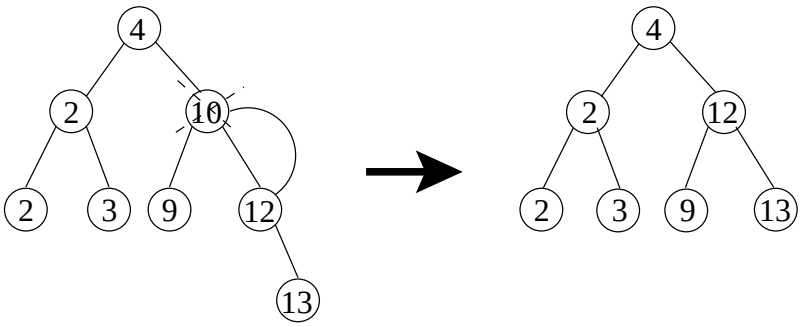

Figura I.11. Rimozione del nodo di chiave 10 dalla'lbero dell'esempio

La complessità di tutte queste operazioni è $O(h)$.

I.5.3 Alberi rosso-neri

Questa tipologia di alberi, in pratica un'estensione degli alberi binari, è stata introdotta per garantirne il bilanciamento, anche a fronte di modifiche alla struttura che una INSERT o una DELETE possono introdurre, senza però appesantire la complessità. Un albero rosso-nero deve rispettare le seguenti regole:

1. Ogni nodo ha un colore, rosso oppure nero.
2. La radice è sempre nera.
3. Le foglie fittizie NIL (strato più basso dell'albero) sono sempre nere.
4. Se un nodo è rosso, allora ha entrambi i figli neri.
5. Per ogni nodo i tutti i cammini da i alle foglie del sottoalbero di radice i hanno lo stesso numero di nodi neri.

L'altezza di un albero rosso-nero è al massimo $2\lg(n+1)$. Questo fa sì che un albero rosso-nero sia un buon albero di ricerca: la ricerca può sempre essere effettuata in $O(\lg n)$, così come le altre operazioni la cui complessitá diipende dall'altezza.

La figura I.12 mostra un esempio di albero rosso-nero; da notare le foglie NIL (rappresentate con un punto nero), sempre presenti in un albero di questo tipo, spesso dette anche 'nodi sentinella', in quanto hanno funzioni particolari negli algoritmi di questi alberi.

Al contrario degli alberi normali, negli alberi rosso-neri le operazioni di INSERT e DELETE sono più complicate, in quanto esse possono infrangere le propriet à dell'albero rosso-nero. Per evitare questo è necessario introdurre alcune operazioni utili.

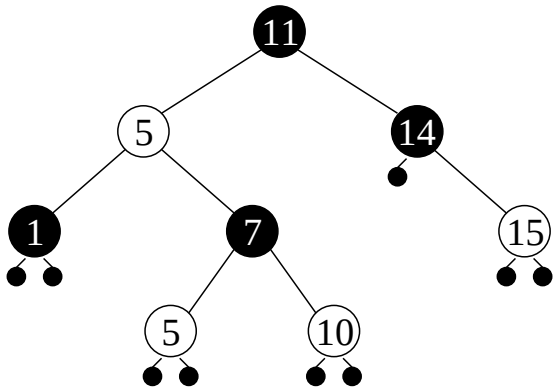

Figura I.12. Esempio di albero rosso-nero

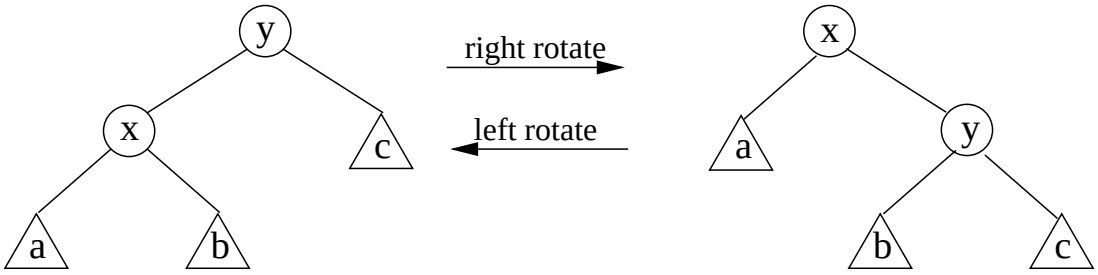

Figura I.13. Rotazione destra e sinistra di un albero binario

Rotazione La rotazione è un'operazione locale che permette di mantenere l'ordinamento delle chiavi.

Una rotazione può essere implementata come mostrato nell'algoritmo RbtLeftRotate.

Insert L'inserzione in un albero rosso-nero può comportare la violazione delle regole 2 e 4. La regola 2 può essere sistemata semplicemente cambiando colore al nodo che è diventato la nuova radice. Per la violazione della regola 4 invece la situazione è più delicata e bisogna valutare diverse alternative.

L'algoritmo funziona fondamentalmente nel seguente modo: si cerca la posizione in cui inserire il nodo, quindi lo si inserisce nella posizione determinata, come avviene per gli alberi binari. Per cercare la posizione si effettuano le seguenti operazioni:

1. Inizializzare il nodo corrente con la radice
2. Confrontare la nuova chiave da inserire con quella del nodo corrente
3. Se è più piccola, scegliere come nuovo nodo corrente il figlio sinistro, altrimenti scegliere il figlio destro
4. Se il nodo corrente è NIL, allora inserire il nuovo nodo nella posizione corrente e terminare l'algoritmo. Altrimenti tornare al punto 2 e ripetere il ciclo

Una volta individuata la posizione in cui inserire il nodo, è necessario controllare il colore dei nodi appartenenti all'albero per verificare eventuali infrazioni della regola 4.

Algoritmo I.11 RBTLEFTROTATE(T, x)

```
 1: y ← right(x)
 2: right(x) ← left(x)
 3: if left(x) ≠ NIL then
 4:    p(left(y)) ← x
 5: p(y) ← p(x)
 6: if p(x) = NIL then
 7:    root(T) = y
 8: else if x = left(y) then
 9:    left(p(x)) ← y
10: else
11:    right(p(x)) ← y
12: left(y) ← x
13: p(x) ← y
```

Algoritmo I.12 RBTINSERT(T, x)

```
 1: Esegui inserzione standard di alberi binari
 2: color(x) ← rosso
 3: while x ≠ root(T) and color(p(x)) = rosso do
 4:    if p(x) = left(p(p(x))) then
 5:       zio(x) ← right(p(p(x)))
 6:       if color(zio(x)) = rosso then
 7:          color(p(x)) ← nero
 8:          color(zio(x)) ← nero
 9:          color(p(p(x))) ← rosso
10:          x ← p(p(x))
11:       else if x = right(x) then
12:          x ← p(x)
13:          RBTROTATELEFT(T,x)
14:          color(p(x)) ← nero
15:          color(p(p(x))) ← rosso
16:          RBTROTATERIGHT(T,p(p(x)))
17:    else
18:       ... caso destro
19: color(root(T)) ← nero
```

Delete Anche la cancellazione può essere fatta in maniera simile a quella dei normali alberi binari. Bisogna però tenere traccia delle conseguenze della cancellazione:

- Se il nodo cancellato era una foglia, nessuna propriet à degli alberi bilanciati viene meno, dal momento che il nodo viene rimpiazzato da una foglia NIL
- Se invece il nodo eliminato viene rimpiazzato da un altro nodo, bisogna controllare se questo nodo è nero, perché in questo caso bisogna apportare alcune correzioni

Algoritmo I.13 RBTDELETE(T, z)

```
1: if left(z) = NIL or right(z) = NIL then
2:     y ← z
3:     while left(y) ≠ NIL do
4:         y ← left(y)
5: if left(y) ≠ NIL then
6:     x ← left(y)
7: else
8:     x ← right(y)
9: p(x) ← p(y)
10: if p(y) ≠ NIL then
11:     if y = left(p(y)) then
12:         left(p(y)) ← x
13:     else
14:         right(p(y)) ← x
15: else
16:     root(T) ← x
17: if y ≠ z then
18:     key(z) ← key(y)
19: if color(y) = nero then
20:     CORREZIONE(T,x)
```

Nella tabella I.2 vengono mostrati alcuni casi esemplificativi di aggiustamento di posizioni e colori in alberi rosso-neri. La posizione z indica il nodo appena aggiunto.

I.6 Intersezione di semipiani e map overlay

Questa applicazione di geometria computazionale, che può ancora una volta essere ricondotta al problema dell'intersezione di segmenti, riguarda la sovrapposizione di 2 mappe, e ha molteplici applicazioni in cartografia, urbanistica e discipline simili. I problemi che stanno alla base di questa applicazione sono l'identificazione di aree (e poligoni) e l'intersezione di semipiani.

Algoritmo I.14 CORREZIONE(T, x)

```
 1: while x ≠ root(T) and color(x) = nero do
 2:    if x = left(p(x)) then
 3:       w ← right(p(x))
 4:       if color(w) = rosso then
 5:          color(w) ← nero
 6:          color(p(w)) ← rosso
 7:          RbtRotateLeft(T,p(x))
 8:          w ← right(p(x))
 9:       if color(right(w)) = rosso and color(left(w)) = rosso then
10:          color(w) ← rosso
11:          x ← p(x)
12:       else if color(right(w)) = nero then
13:          color(left(w)) ← nero
14:          color(w) ← rosso
15:          RbtRotateRight(T,w)
16:          w ← right(p(x))
17:       color(w) ← color(p(x))
18:       color(p(x)) ← nero
19:       color(right(w)) ← nero
20:       RbtRotateLeft(T,p(x))
21:       x ← root(T)
22:    else
23:       stessa porzione di codice con left e right invertiti
24: color(x) ← nero
```

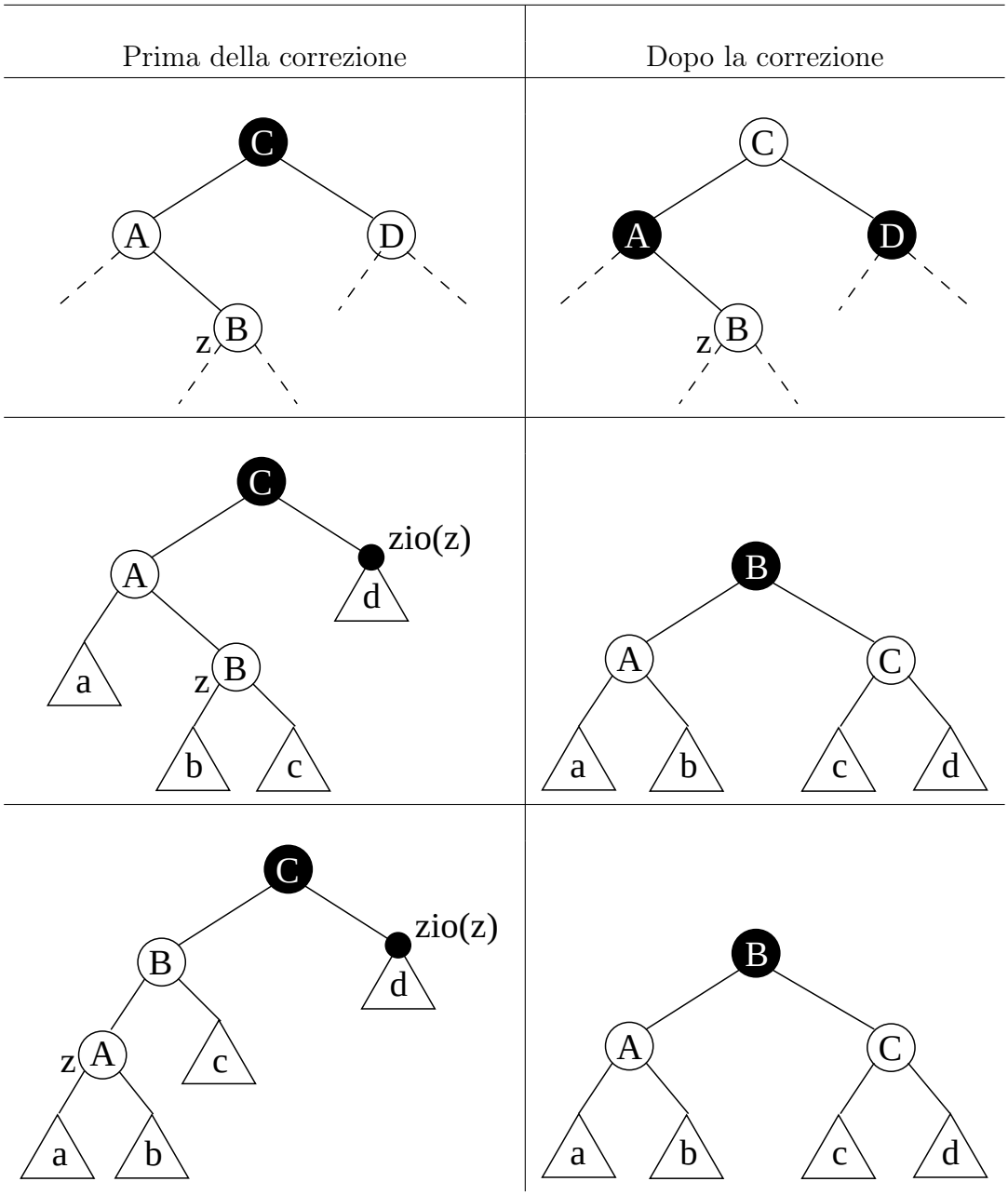

Tabella I.2. Esempi di correzione di alberi colorati

I.6.1 Costruzione di poligoni

Un poligono semplice (senza 'buchi' al suo interno) può essere rappresentato da una doppia lista linkata, corrispondente ai due percorsi interno ed esterno, dove per ogni punto è indicato il successore e il predecessore.

Dato un poligono si identificano il percorso esterno e quello interno (per convenzione il contenuto del poligono sta a sinistra del segmento orientato). Dato un punto p (nella figua I.15 corrispondente a $origin(e)$), si definiscono i seguenti elementi:

- e: segmento uscente da p

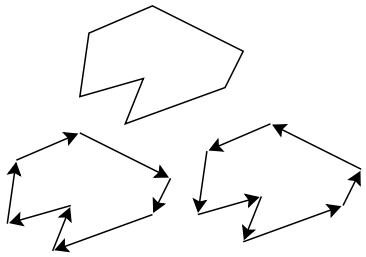

Figura I.14. Esempio di poligono e di percorso esterno e interno

- $twin(e)$: segmento entrante in p
- $prev(e)$: predecessore di e, entrante anch'esso in p
- $next(e)$: successore di e.

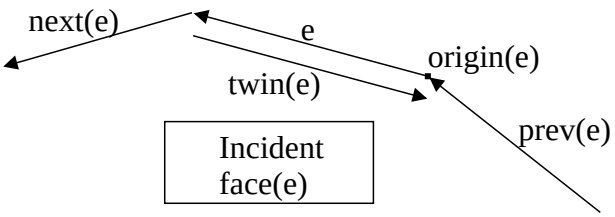

Figura I.15. Elementi base necessari per descrivere un poligono

Per memorizzare un poligono bisogna tenere conto di questi elementi. Sono necessarie tre distinte strutture dati. Per presentarle facciamo uso dell'esempio seguente.

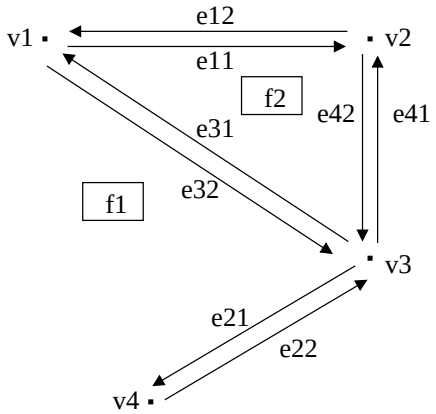

Figura I.16. Poligono di esempio con quattro vertici

Esempio Dato il poligono in figura I.16, le strutture dati necessarie sono presentate di seguito.

Nella sovrapposizione di mappe, una volta determinati i punti di intersezione tra i lati delle varie regioni, utilizzando l'algoritmo visto in precedenza, il problema consiste nell'aggiornare la struttura dati dei vertici, delle facce e dei lati.

Prima struttura dati: vertici

	Coordinate	IncidentEdge
$v1$	...	$e11$
$v2$	...	$e42$
$v3$	...	$e21$
$v4$	...	$e22$

Seconda struttura dati: facce

	OuterComponent	InnerComponent
$f1$	NIL	$e11(\ldots)$
$f2$	$e41$	NIL

Terza struttura dati: lati

	Origin	Twin	IncidentFace	Next	Prev
$e11$	$v1$	$e12$	$f1$	$e42$	$e31$
$e12$	$v2$	$e11$	$f2$	$e32$	$e41$
$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$	$\vdots$

I.6.2 Intersezione di semipiani

Considerando un insieme di semipiani $H = \{h_1, \ldots, h_n\}$, ci si pone il problema di individuare il poligono convesso definito dall'intersezione dei vari semipiani.

Un modo di risolvere il problema fa uso dello schema dividi et impera. L'insieme H viene suddiviso in due sottoinsiemi di pari cardinalitá e il problema viene risolto sui due sottoinsiemi per poi ricomporre la soluzione facendo l'intersezione dei due poligoni.

L'algoritmo ha una complessit à che dipende dalla ricorsione, pertanto $T(n) = O(n \lg n) + 2T(n/2)$ dove la prima parte è dovuta all'intersezione, la seconda parte alla ricorsione (suddivisione del problema in 2 sottoproblemi secondo l'approccio divide et impera). Applicando il Master Theorem si ricava una complessit à di $O(n \lg^2 n)$.

Per migliorare le prestazioni, si possono richiamare i concetti introdotti per l'intersezione di segmenti: si definisce una *sweep line* e si cerca tutte le linee (bordi dei semipiani) che la sweep line interseca. Essendo le figure convesse, al massimo la sweep line intersecher à 4 linee (considerando che per ogni lato vengono definiti 2 segmenti orientati). Per questo motivo, essendoci soltanto 4 variabili in gioco, invece di utilizzare strutture dati complesse, basta definire 4 variabili $c1$, $c2$, $c3$, $c4$ che memorizzano la coordinata y di intersezione dei segmenti con la sweep line. In questo modo, usando la sweep line e l'approccio divide et impera, si riduce la complessit à a $O(n \lg n)$.

I.6.3 Videosorveglianza di un museo

In questo paragrafo viene presentata un'applicazione pratica dei problemi precedentemente descritti. Dato un museo, suddiviso in stanze, si vuole posizionare un certo numero di telecamere in modo da sorvegliare tutte le stanze. Il problema è quello di determinare un limite superiore al numero di telecamere necessarie (si ricorda che il problema di determi-

Algoritmo I.15 IntersectHalfplanes(H)

```
1: H = insieme dei semipiani
2: if |H| = 1 then
3:    c ← h ∈ H
4: else
5:    Dividi H in H1 e H2 con |H1| = ⌈n/2⌉ e |H2| = ⌊n/2⌋
6:    C1 ← IntersectHalfplanes(H1)
7:    C2 ← IntersectHalfplanes(H2)
8:    C ← IntersectConvexRegions(C1,C2)
9: return(C)
```

nare invece il numero minimo di telecamere utili è un problema NP-hard). Per risolvere il problema, bisogna ricondurre la struttura del museo a quella di un poligono semplice.

Esempio Dato un certo museo, esso viene ridotto ad un poligono semplice (vedi figura I.17), nell'esempio il poligono ottenuto ha $n = 14$ vertici.

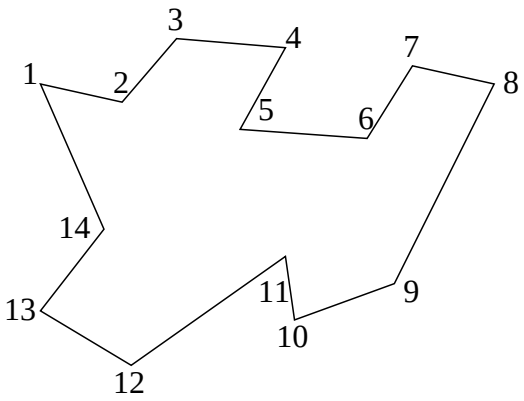

Figura I.17. Poligono rappresentante i contorni di un museo

Una prima possibilit à è quella di mettere una telecamera in ogni vertice. In questo modo si deduce che servono $n = 14$ telecamere. Si ha un evidente spreco di risorse, in quanto una telecamera potrebbe essere utilizzata per coprire più stanze contemporaneamente.

L'idea alternativa è quella di decomporre la figura in triangoli. Un poligono semplice è sempre decomponibile in $n - 2$ triangoli.

È possibile quindi mettere una telecamera su ogni riga che è stata tracciata per dividere il poligono. In questo modo, essendo ogni lato comune a due triangoli, ed essendoci $n - 2$ triangoli, sono necessarie $(n - 2)/2 = 6$ telecamere. è comunque possibile fare ancora meglio. La tecnica da utilizzare è la colorazione dei vertici con tre colori diversi. La regola da seguire è quella per cui vertici adiacenti (cioè collegati da un lato) devono avere colori differenti. Questo implica che i tre vertici di ogni triangolo hanno tre colori differenti.

In questo modo è sufficiente porre una telecamera su tutti i vertici di un medesimo colore. Ad esempio, posizionando le telecamere solo sui vertici neri, verranno messe

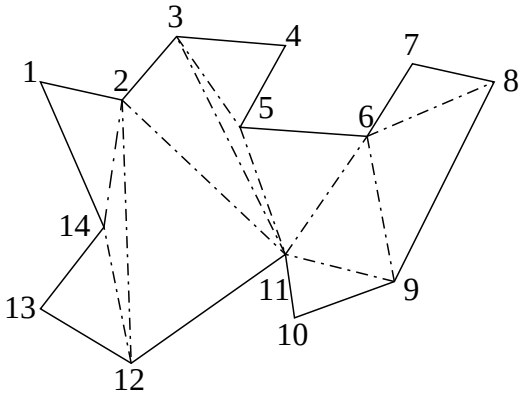

Figura I.18. Poligono della figura I.17 suddiviso in triangoli

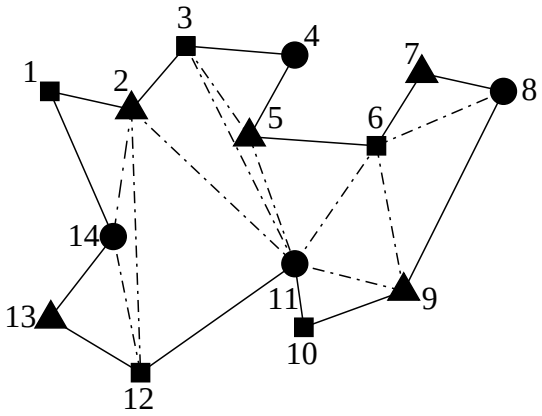

Figura I.19. Poligono della figura I.18 con i vertici suddivisi in tre categorie

$n/3 = 4$ telecamere nei vertici 4, 8, 11 e 14. È stato dimostrato (in due differenti modi, che non tratteremo) che 3 colori sono sufficienti per risolvere questo problema.

I.7 Diagrammi di Voronoi

Il problema che sta alla base della teoria dei diagrammi di Voronoi, introdotti nel 1908 da Voronoi e successivamente studiati con un trattazione algoritmica a partire dal 1965, si trova gi à nelle opere di Cartesio, nel XVII secolo. La questione riguarda, dato un insieme di punti $P = \{p_1, \ldots, p_n\}$ distribuiti nel piano, suddividere il piano in n celle, ognuna contenente uno dei punti p_i. Formalmente, per costruire un diagramma di Voronoi, sono necessari i seguenti elementi:

- $P = \{p_1, \ldots, p_n\}$: insieme dei punti p_i
- $Dist(p, q) = \sqrt{(x_p - x_q)^2 + (y_p - y_q)^2}$: distanza (euclidea) dei punti p e q
- $V(p_i)$: cella i-esima, contenete il punto $p_i, i = 1, \ldots, n$
- $Vor(P)$: diagramma di Voronoi, cioè suddivisione del piano in n celle, ciascuna contenente un punto $p_i, i = 1, \ldots, n$

Un punto $q \in V(p_i)$ se $Dist(q, p_i) \leq Dist(q, p_j)$ con $i \neq j$.

I.7.1 Costruzione del diagramma di Voronoi con la definizione geometrica

Cerchiamo di caratterizzare le celle di un diagramma di Voronoi. Dati due punti p_i e p_j nel piano, vediamo come determinare il confine (se esiste) tra $V(p_i)$ e $V(p_j)$.

Sia $\overline{p_i p_j}$ il segmento che unisce p_i e p_j e $bisector(\overline{p_i p_j})$ l'asse del segmento. *bisector* suddivide il piano in due semipiani, $h(p_i, p_j)$ e $h(p_j, p_i)$, e i punti sull'asse sono equidistanti da p_i e da p_j. La cella $V(p_i)$ si può ottenere facendo

$$V(p_i) = \bigcap_{i \neq j} h(p_i, p_j)$$

utilizzando l'algoritmo di intersezione di semipiani visto prima.

Questo procedimento può essere formalizzato in un algoritmo per la costruzione di $Vor(P)$. Si noti come i lati di ogni cella risultino essere rettilinei.

Algoritmo I.16 VoronoiConstruction(P)

1: **for each** $(p_i, p_j), i < j$ **do**
2: costruisci segmento $\overline{p_i p_j}$
3: costruisci $bisector(\overline{p_i p_j})$
4: determina semipiani $h(p_i, p_j)$ e $h(p_j, p_i)$
5: **for each** p_i **do**
6: $V(p_i) = \bigcap_{i \neq j} h(p_i, p_j)$

Si può dimostrare che ha una complessità di $O(n^2 \lg n)$, infatti dobbiamo fare n intersezioni di $n - 1$ semipiani.

Cerchiamo di caratterizzare meglio i diagrammi di Voronoi, in particolare cercando di valutare il numero di vertici e di lati. Associamo il diagramma di Voronoi ad un grafo planare (ossia senza intersezione dei lati) dove ciascun punto p_i rappresenta una faccia del grafo, e ciascun lato del diagramma rappresenta un lato del grafo. Bisogna aggiungere un vertice fittizio v_{inf} a cui far convergere le semirette (lati infiniti) delle facce aperte di $Vor(P)$.

Questa trattazione può essere fatta per un diagramma di Voronoi generico, ad esclusione del caso particolare in cui invece di lati e semipiani, la costruzione di $Vor(P)$ d à origine a una successione di rette parallele (questo accade se i punti p_i sono allineati). Dato un grafo planare, si possono definire le seguenti grandezze:
m_V = numero vertici del grafo
m_L = numero lati del grafo
m_F = numero facce del grafo
Ricordiamo la formula di Eulero: $m_V - m_L + m_F = 2$.

Nel grafo costruito per $Vor(P)$:
$m_V = n_V + 1$, dove al numero di vertici n_V va aggiunto il vertice v_{inf}
$m_L = n_L$
$m_F = n_F$
Considerato che:

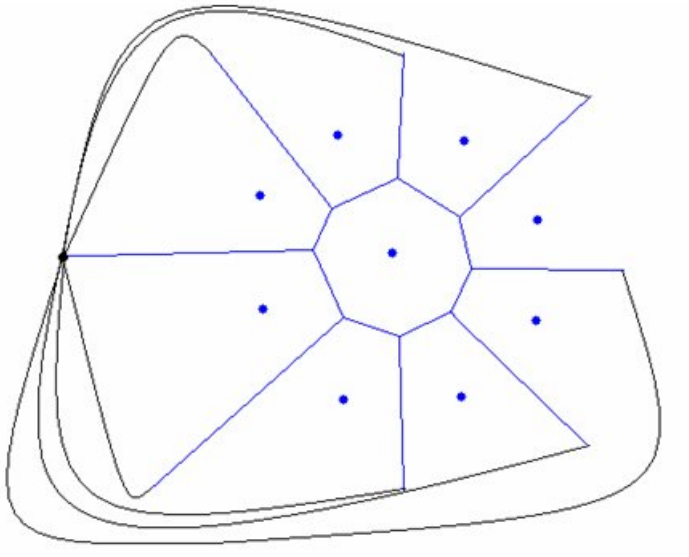

Figura I.20. Grafo planare

- Ogni lato ha 2 vertici (infatti abbiamo aggiunto v_{inf})
- Ogni vertice ha almeno tre lati incidenti
- La somma dei lati incidenti di tutti i vertici del grafo eguaglia il doppio del numero dei lati del grafo, ossia

$$2n_L \geq 3(n_V + 1)$$

Sviluppando la formula di Eulero si ricava (dove $n = n_F$):
$n_L \leq 3n - 6$
$n_V \leq 2n - 5$

Quindi il numero di vertici e il numero dei lati di un diagramma di Voronoi dipendono linearmente dal numero di punti di P.

I.7.2 Algoritmo Fortune

Questo algoritmo permette di costruire il diagramma di Voronoi in maniera più efficiente e veloce, con un tempo di computazione pari a $O(n \lg n)$, che risulta essere un limite inferiore poichè è stata provata l'equivalenza tra la costruzione di un diagramma di Voronoi e l'ordinamento di n numeri. Vengono sfruttate due proprietà fondamentali dei diagrammi di Voronoi.

Proprietà dei vertici Un punto q è un vertice di $Vor(P)$ se si può definire il cerchio $C_P(q)$ con centro in q, che contiene 3 o più punti $p_i \in P$ sul bordo e nessun punto di P al suo interno.

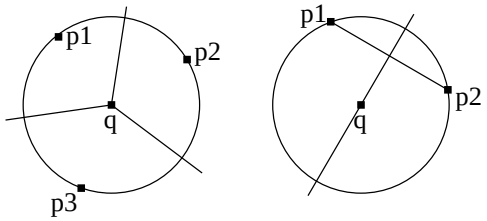

Figura I.21. Proprietà dei vertici (a sinistra) e proprietà dei lati (a destra)

Proprietà dei lati L'asse del segmento $\overline{p_i p_j}$ è un lato di $Vor(P)$ se esiste un punto $q \in bisector(p_i p_j)$ tale che è possibile costruire il cerchio $C_P(q)$ con centro in q e che contiene i punti p_i e p_j sul bordo.

L'algoritmo FORTUNE si basa su una *sweep line*, che muovendosi da sinistra verso destra scandisce tutti i punti p_i del piano e in ogni istante verifica se le proprietà sopra enunciate sono valide. In caso affermativo costruisce in maniera progressiva il diagramma di Voronoi.

Il punto delicato dell'algoritmo sta proprio nella sua progressività. Muovendosi verso destra nel tempo, in ogni istante si conosce tutto quanto è a sinistra della sweep line, ma non si conosce nulla di quanto sta a destra di essa. Questo impedisce di conoscere come i lati già disegnati del diagramma di Voronoi evolveranno mano a mano che verranno trovati nuovi punti dalla sweep line. è pertanto necessario caratterizzare i punti a sinistra della sweep line la cui attribuzione alle celle non è influenzata dai nuovi punti a destra della stessa. Per ognuno di questi punti viene definita in modo dinamico una parabola, che delimita la zona in cui i nuovi punti che verranno scoperti non hanno influenza da quella in cui essi possono apportare modifiche alla struttura dei lati di $Vor(P)$. L'unione di tutte queste parabole va a definire una linea spezzata di frontiera, che per la sua forma (che richiama la linea mossa delle spiagge e delle coste) viene chiamata *beach line*. Tutto ciò che sta a sinistra della beach line rappresenta dati ormai consolidati e non più modificabili, ossia il diagramma di Voronoi già disegnato alla sua sinistra è definitivo.

Una beach line è caratterizzata dalle seguenti proprietà:

1. Ciascuna parabola associata a un punto può intervenire più di una volta a comporre diverse parti della spezzata

2. Per i break-point della beach line passano i lati del diagramma di Voronoi

L'algoritmo FORTUNE, nella costruzione della beach line, deve tenere conto di due eventi fondamentali:

Creazione di un nuovo lato (site event) Quando la sweep line raggiunge un nuovo punto. Una volta che viene incontrato un nuovo punto viene creata una nuova parabola (all'inizio degenere in un segmento) che andrà ad aprirsi e ad integrarsi nella beach line. Il nuovo lato è creato. Il nuovo lato evolverà assieme alla beach line e sarà stabile quando si verifica il circle event.

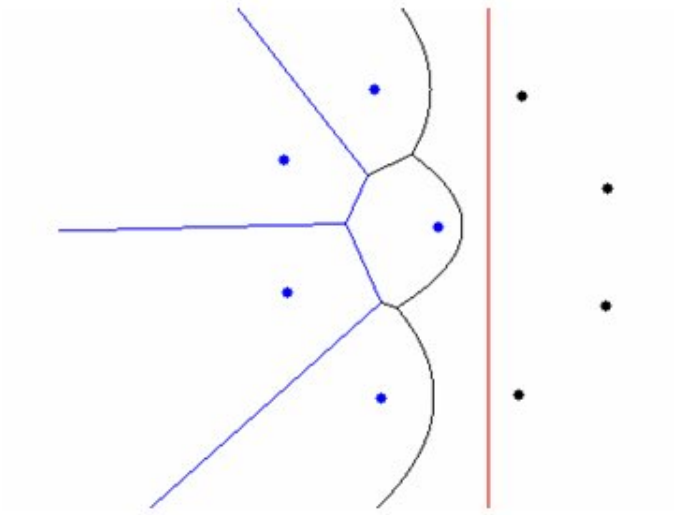

Figura I.22. Costruzione del diagramma di Voronoi effettuata con l'algoritmo FORTUNE

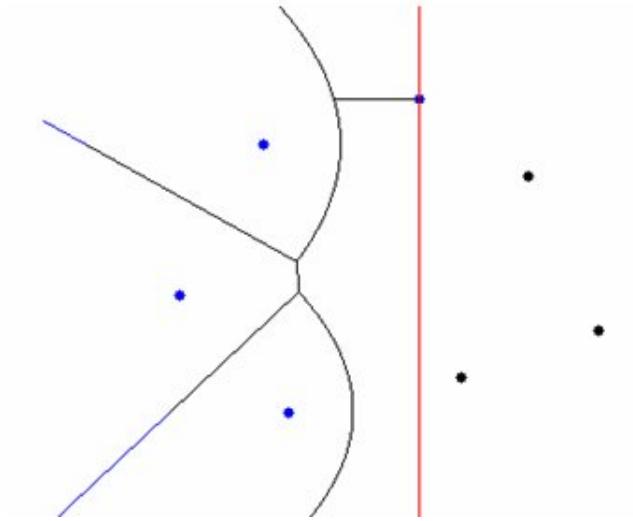

Figura I.23. Creazione di un nuovo lato (site event)

Creazione di un nuovo vertice (circle event) Avviene quando sono verificate le condizioni per la creazione di un cerchio. Al generarsi di un nuovo evento il vertice diventa stabile, e il cerchio viene rimosso.

L'algoritmo, per essere efficiente, deve utilizzare alberi rosso-neri come struttura dati per la rappresentazione della beach line (dove come chiave viene memorizzata la coordinata y del punto p_i), e un albero o una coda di priorità per la event queue. In questo modo si può garantire una complessità di $O(n \lg n)$.

È possibile visualizzare una demo dell'algoritmo sul sito http://www.diku.dk/hjemmesider/studerende/duff/Fortune/

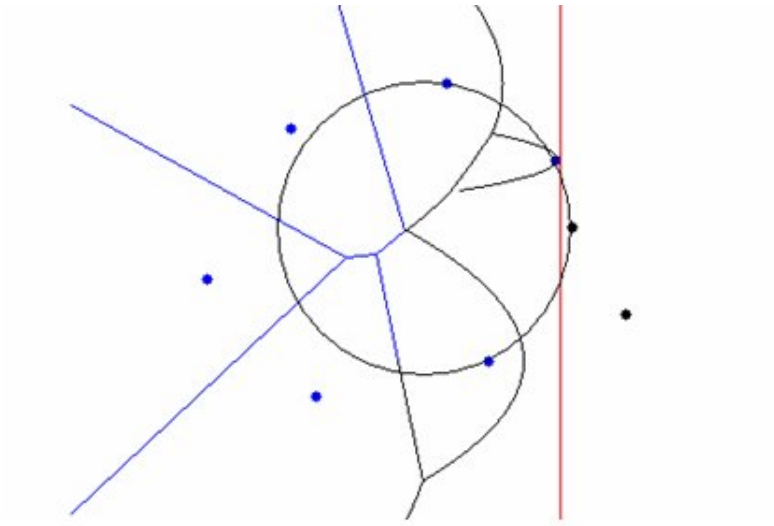

Figura I.24. Creazione di un nuovo vertice (circle event)