Computabilità ed Algoritmi
Esame di Algoritmi
18 Giugno 2018

Tempo a disposizione: 1 h 30 min

1. *Flappy Bird* è un gioco per cellulari dove l'obiettivo è quello di far volare un uccello chiamato "Faby" attraverso una serie di tubi, evitando di farlo scontrare con essi o di farlo cadere per terra. Faby vola verso destra a velocità costante. Ogni volta che il giocatore tocca lo schermo, a Faby viene data una velocità ascendente fissa; altrimenti, se non riceve comandi Faby cade verso il fondo per gravità.

In questo esercizio si chiede di sviluppare un algoritmo per giocare automaticamente alla seguente variante semplificata di Flappy Bird, chiamata *Flappy Pixel*. Invece di un uccello, Faby è un singolo punto, specificato da tre numeri interi: posizione orizzontale x (in pixel), posizione verticale y (in pixel) e velocità verticale y' (in pixel per fotogramma). L'ambiente di Faby è descritto da due array $Hi[0 \dots n-1]$ e $Lo[0 \dots n-1]$, dove per ogni indice i, abbiamo $0 < Lo[i] < Hi[i] < h$ per un determinato valore di altezza h. Il gioco è descritto dal seguente pezzo di pseudocodice:

```
function FlappyPixel(Hi[0...n−1], Lo[0...n−1])
    y = h/2
    y' = 0
    for x = 0 to n − 1 do
        if il giocatore tocca lo schermo then
            y' = 10
        else
            y' = y' − 1
        y = y + y'
        if y < Lo[x] or y > Hi[x] then
            return False                ▷ Il giocatore ha perso
    return True                         ▷ Il giocatore ha vinto
```

Si noti che in ogni iterazione del ciclo principale, il giocatore ha la possibilità di toccare lo schermo.

Considerate il problema di determinare il numero *minimo* di volte in cui il giocatore deve toccare lo schermo per vincere Flappy Pixel, dato l'intero h e gli array $Hi[0 \dots n-1]$ e $Lo[0 \dots n-1]$ come input. Trasformate l'input in un grafo e applicate uno degli algoritmi visti a lezione per risolverlo.

Modelliamo il problema con un grafo diretto e pesato $G = (V, E)$:

- I vertici di G sono triple (x, y, y') che rappresentano il valore delle tre variabili che descrivono lo stato di Faby: posizione orizzontale e verticale e velocità; il grafo comprende solo le triple dove la coordinata y rispetta le condizioni date dai vettori Hi e Lo: $Lo[x] \leq y \leq Hi[x]$.
- Gli archi corrispondono alle mosse valide:
  - da (x, y, y') verso $(x+1, y+10, 10)$ (il giocatore tocca lo schermo);
  - da (x, y, y') verso $(x+1, y+y'-1, y'-1)$ (il giocatore non tocca lo schermo).
- I vertici sono associati allo stato di Faby come specificato. Gli archi sono pesati: peso 1 per gli archi dove il giocatore tocca lo schermo, 0 per gli archi dove il giocatore non tocca lo schermo.
- Il problema da risolvere è trovare il cammino minimo da $(0, h/2, 0)$ verso un vertice $(n-1, y, y')$ dove il giocatore vince.
- L'algoritmo da usare per risolvere il problema è un qualsiasi algoritmo per il problema del cammino minimo da sorgente singola, come per esempio l'algoritmo di Dijkstra.
- Il tempo impiegato per costruire il grafo è proporzionale alla sua dimensione. Il numero dei vertici dipende dai valori che le variabili x, y, y' possono assumere. x varia da 0 a $n-1$, y da 0 a h e y' tra $-h/2$ e 10. Da ogni vertice partono al più due archi, quindi il grafo ha $n \cdot h \cdot (h/2+10)$ vertici e al più $2(n \cdot h \cdot (h/2+10))$ archi. La sua costruzione ha costo asintotico $O(h^2 n)$. La complessità dell'algoritmo di Dijkstra è $O(|E| + |V| \log |V|) = O(h^2 n \log(h^2 n))$. Quindi l'algoritmo complessivamente impiega tempo $O(h^2 n \log(h^2 n))$.

2. Supponiamo di avere un grafo pesato e non orientato G e un albero di copertura minimo T di G.

(a) Descrivere un algoritmo per aggiornare l'albero di copertura minimo quando il peso di un singolo arco $\{u, v\}$ viene *diminuito*.

> Se l'arco $\{u, v\} \in T$ allora T continua ad essere un albero di copertura minimo e l'algoritmo restituisce T senza modificarlo.
> Se l'arco $\{u, v\} \notin T$ allora l'algoritmo procede come segue:
> - aggiunge l'arco $\{u, v\}$ a T; questa operazione crea un unico ciclo in T che comprende anche l'arco $\{u, v\}$
> - tale ciclo può essere trovato con una visita in profondità di T che parte da u, che ha un costo $O(|V|)$ (il numero di archi in T è $O(|V|)$)
> - il nuovo albero di copertura minimo si ottiene eliminando l'arco di peso massimo presente nel ciclo, operazione che costa $O(|V|)$.
>
> La complessità totale dell'algoritmo è $O(|V|)$.

(b) Descrivere un algoritmo per aggiornare l'albero di copertura minimo quando il peso di un singolo arco $\{u, v\}$ viene *aumentato*.

> Se l'arco $\{u, v\} \notin T$ allora T continua ad essere un albero di copertura minimo e l'algoritmo restituisce T senza modificarlo.
> Se l'arco $\{u, v\} \in T$ allora l'algoritmo procede come segue:
> - toglie l'arco $\{u, v\}$ da T; questa operazione divide T in due sottoalberi T_1 e T_2;
> - T_1 e T_2 definiscono un taglio che li rispetta. Per il teorema visto a lezione, ogni arco leggero che attraversa il taglio è un arco sicuro per T_1 e T_2.
> - il nuovo albero di copertura minimo si ottiene collegando T_1 a T_2 con un arco di peso minimo che attraversa il taglio.
> - T_1 e T_2 possono essere determinati con due visite in profondità di $T \setminus \{\{u, v\}\}$, una che parte da u e una che parte da v. Il costo delle due visite è $O(|V|)$.
> - Per trovare l'arco leggero è sufficiente scorrere tutti gli archi di G, considerando solo quelli che hanno un estremo in T_1 e l'altro in T_2 e determinare quello di peso minimo. Questa operazione ha un costo asintotico $O(|E|)$.
>
> L'algoritmo ha complessità asintotica $O(|V| + |E|)$.

(c) Giustificare la correttezza degli algoritmi proposti.

> Vedi le soluzioni dei punti (a) e (b).

(d) Determinare la complessità asintotica degli algoritmi proposti.

> Vedi le soluzioni dei punti (a) e (b).

In tutti i casi, gli input dell'algoritmo sono G, T, l'arco $\{u, v\}$ e il suo nuovo peso. Gli algoritmi devono modificare T in modo che sia ancora un albero di copertura minimo. Ovviamente, si potrebbe semplicemente ricalcolare da zero l'albero di copertura minimo in tempo $O(|E| + |V| \log |V|)$, ma si può fare di meglio.

3. Dimostrare che un problema X è NP-hard richiede diversi passaggi:

- Scegli un problema Y che sai essere NP-hard.
- Descrivi un algoritmo per risolvere Y usando un algoritmo per X come subroutine. Tipicamente questo algoritmo ha la seguente forma: data un'istanza di Y, trasformala in un'istanza di X, quindi chiama l'algoritmo magico black-box per X.
- Dimostra che l'algoritmo è corretto. Ciò richiede sempre due passaggi separati, che di solito hanno la seguente forma:
  - Dimostra che il tuo algoritmo trasforma istanze "buone" di Y in istanze "buone" di X.
  - Dimostra che il tuo algoritmo trasforma istanze "cattive" di Y in istanze "cattive" di X. Equivalentemente: Dimostra che se la tua trasformazione produce un'istanza "buona" di X, allora era partita da un'istanza "buona" di Y.
- Mostra che il tuo algoritmo per Y funziona in tempo polinomiale, a meno della chiamata (o delle chiamate) all'algoritmo magico black-box per X. (Questo di solito è banale.)

"Colorare" i vertici di un grafo significa assegnare etichette, tradizionalmente chiamate "colori", ai vertici del grafo in modo tale che nessuna coppia di vertici adiacenti condivida lo stesso colore. Il problema kCOLOR è il problema di trovare una colorazione di un grafo non orientato usando k colori diversi. Sappiamo che il problema 3COLOR è NP-Hard.

(a) Descrivete un algoritmo per risolvere 3COLOR usando 4COLOR come subroutine.

> Sia G il grafo non orientato di input per 3COLOR. L'algoritmo crea un nuovo grafo H aggiungendo un nuovo vertice v_0 al grafo G, collegato con ogni altro vertice di G mediante un arco. Quindi si esegue 4COLOR su H, ritornando la risposta ottenuta come risultato finale.
>
> Dimostriamo che esiste un modo per colorare G con 3 colori se solo se esiste un modo per colorare H con 4 colori:
>
> - Supponiamo che esista un modo per colorare G con 3 colori "rosso", "verde" e "blu". Assegnamo al nuovo vertice v_0 un quarto colore, per esempio "giallo". Consideriamo un arco $\{u, v\}$ di H:
>   - se u e v sono entrambi vertici di G, allora hanno un colore diverso;
>   - altrimenti, uno dei due estremi dell'arco è v_0, che è colorato di giallo, e l'altro si trova in G, e quindi può essere colorato di rosso, verde o blu. In tutti i casi i colori dei due vertici sono diversi.
>
>   Possiamo concludere che abbiamo trovato un modo per colorare H con 4 colori.
> - Supponiamo che H sia colorabile con 4 colori. Chiamiamo "giallo" il colore assegnato al nuovo vertice v_0, e "rosso", "verde" e "blu" gli altri tre. Ogni vertice u in G è collegato da un arco con v_0, quindi non può essere colorato di giallo. Di conseguenza, tutti i vertici di G sono colorati di "rosso", "verde" o "blu" in modo che gli estremi di ogni arco siano di colore diverso. Questo vuol dire che esiste un modo per colorare G con tre colori.

(b) Dimostrare che kCOLOR è NP-Hard per qualsiasi $k \geq 3$.

> Sappiamo già che kCOLOR è NP-Hard quando $k = 3$. Per dimostrare che è NP-Hard anche per $k > 3$ è sufficiente generalizzare l'algoritmo proposto al punto precedente per un numero di colori arbitrario k. Se G è un grafo non orientato di input per 3COLOR, l'algoritmo crea il grafo H aggiungendo $k - 3$ nuovi nodi $v_0, \ldots, v_{k-4}$ e collegando ogni nuovo vertice a tutti i vertici di G e con tutti gli altri vertici v_i. Quindi esegue kCOLOR sul nuovo grafo.
>
> In maniera analoga a quanto visto sopra, possiamo dimostrare che G è colorabile con 3 colori se e solo se H è colorabile con k colori. La costruzione del grafo H richiede un tempo polinomiale, e quindi abbiamo dimostrato che kCOLOR è NP-Hard.