6 Esercizi

6.1 Costruzione di semi-anelli

Supponiamo fissato un semi-anello A, non necessariamente chiuso, ed una sua implementazione (per semplicita' si supponga che le implementazioni delle operazioni su A abbiano costo costante).

Semi-anello delle funzioni. Dato un qualunque insieme X, l'insieme delle funzioni $X \to A$ ha una struttura di semi-anello ottenuta estendendo in modo puntuale le operazioni su A, cioe'

- $f_1 + f_2$ e' la funzione t.c. $(f_1 + f_2)(x) = f_1(x) + f_2(x)$
- $f_1 * f_2$ e' la funzione t.c. $(f_1 * f_2)(x) = f_1(x) * f_2(x)$

In generale il semi-anello $X \to A$ non e' implementabile (perche' le funzioni sono oggetti infinitari, quando X e' un insieme infinito).

Semi-anello dei polinomi in x. Si consideri il semi-anello $A[x]$ dei polinomi in una variabile, un polinomio puo' essere descritto come una somma formale di monomi $p(x) = \sum_{i=0}^{n} a_i * x^i$ con $a_n \neq 0$, n e' detto il grado del polinomio p. Ad ogni polinomio resta associata una funzione $A \to A$, e le operazioni $+$ e $*$ sui polinomi (come somme formali) sono consistenti con quelle definite su $A \to A$.
Implementare il tipo di dato $A[x]$, supponendo di disporre di una implementazione di A, ed analizzare gli algoritmi per le operazioni del semi-anello $A[x]$ in funzione del grado.
OSSERVAZIONE: in alcuni semi-anelli, p.e. quello dei reali, un polinomio di grado $\leq n$ e' univocamente determinato dai valori che assume in $n+1$ punti distinti $x_0, \ldots, x_n$. In questi casi si puo' adottare una rappresentazione alternativa dei polinomi. Analizzare il costo (in funzione del grado del polinomio) per passare da una rappresentazione all'altra.

Semi-anello generato dalle matrici $X \times X$ quasi ovunque nulle. Quando X e' infinito, sulle matrici $X \times X$ quasi ovunque nulle e' possibile definire le operazioni $+$ e $*$, ma l'unita' per $*$, cioe' la matrice identita' I non e' quasi ovunque nulla. Si estenda l'insieme delle matrici $X \times X$ quasi ovunque nulle in modo da ottenere un semi-anello implementabile.
SUGGERIMENTO: si faccia vedere che le matrici della forma $a * I + M$ con $a \in A$ e M matrice quasi ovunque nulla, formano un semi-anello. Implementare tale semi-anello, ed analizzare gli algoritmi per le operazioni in funzione del numero di coefficienti non nulli di M.

6.2 Estensioni del tipo di dato set

Si consideri il tipo di dato set (dove gli elementi hano un ordine lineare) con la seguente segnatura e relativo modello canonico

- sorts – $elem, set, bool$
- ops – leq: $elem, elem \to bool$, $empty$: set, $member$: $elem, set \to bool$,
  ins, del: $elem, set \to set$, $union, intersect$: $set, set \to set$.
- modello canonico A: $elem^A = U$, leq^A ordine lineare $\leq$ su U,
  $set^A = \mathcal{P}(U)$ cioe' l'insieme dei sottoinsiemi di U (non necessariamente finiti)

Operazione differenza. Si introduca una nuova operazione $diff$: $set, set \to set$ con $diff^A(X,Y) = \{x \in X | x \notin Y\}$. Si dica come implementare tale operazione (ed analizzarne la complessita') nelle varie rappresentazioni (parziali) degli insiemi finiti gia' viste a lezione (cioe' bit-vector, liste/array senza ripetizione/ordinati).

Operazione complemento. Si introduca una nuova operazione $compl$: $set \to set$ con $compl^A(X) = \{x \in U | x \notin X\}$. Quando U e' infinito tale operazione fa' uscire da dall'insieme dei sottoinsiemi finiti di U, piu' precisamente occorre rappresentare sia i sottoinsiemi finiti che quelli co-finiti di U ($X \subseteq U$ e' detto co-finito se il suo complementare e' finito). Proporre delle rappresentazioni di tale collezione di sottoinsiemi di U, dire come implementare le operazioni della segnatura, ed analizzare la complessita' delle operazioni (in funzione di parametri opportuni).
SUGGERIMENTO: definire una implementazione B dei sottoinsiemi finiti e co-finiti parametricamente in una implementazione C dei soli sottoinsiemi finiti. L'idea chiave e' di definire $set^B = bool \times set^C$.

Operazione intervallo. Si introduca una nuova operazione $interval: elem, elem \to set$ con $interval^A(x,y) = [x,y] \stackrel{\Delta}{=} \{z \in U | x \leq z \leq y\}$. Quando $(U, \leq)$ e' ordine lineare denso (cioe' se $x < y$, allora esiste uno z t.c. $x < z < y$), un intervallo limitato puo' contenere un numero infinito di punti. Proporre delle rappresentazioni della collezione di sottoinsiemi di U che si ottiene applicando le operazioni della segnatura (ma non *compl*), dire come implementare le operazioni della segnatura, ed analizzare la complessita' delle operazioni (in funzione di parametri opportuni).
SUGGERIMENTO: i sottoinsiemi di U che ci interessano sono descrivibili come unioni finite di intervalli limitati (ma non necessariamente chiusi). Si consiglia di trovare una rappresentazione univoca di tali sottoinsiemi, quindi si dovrebbero evitare intervalli che si intersecano o la cui unione forma un intervallo.

Operazioni complemento ed intervallo. Si consideri l'estensione del tipo di dato set con entrambe le operazioni *compl* e *interval*. Proporre delle rappresentazioni della collezione di sottoinsiemi di U che si ottiene applicando le operazioni della segnatura, dire come implementare le operazioni della segnatura, ed analizzare la complessita' delle operazioni (in funzione di parametri opportuni).
SUGGERIMENTO: i sottoinsiemi di U che ci interessano sono descrivibili come unioni finite di intervalli non necessariamente limitati, come p.e. $(, x] \stackrel{\Delta}{=} \{z \in U | z \leq x\}$ e $[x,) \stackrel{\Delta}{=} \{z \in U | x \leq z\}$. Si consiglia di trovare una rappresentazione univoca di tali sottoinsiemi, quindi si dovrebbero evitare intervalli che si intersecano o la cui unione forma un intervallo. Un approccio alternativo e' basato sulla rappresentazione della funzione caratteristica di un tale sottoinsieme, che avra' necessariamente un numero finito di punti di discontinuita'.

Insiemi ereditari. Si consideri l'estensione del tipo di dato set con una operazione $include: set \to elem$ che permette di vedere un insieme come un elemento. A livello di modello canonico $include^A$ dovrebbe corrispondere all'inclusione $\mathcal{P}_{fin}(U) \subseteq U$, quindi l'interpretazione di *elem* non puo' essere scelta in modo indipendente dalla interpretazione di *set*, ovvero l'interpretazione delle due sort e' mutuamente ricorsiva. Quello che si puo' fare e' definire U in modo induttivo, a partire da un insieme U_0 di elementi (che non sono insiemi):

- $U_{n+1} = U_0 + \mathcal{P}_{fin}(U_n)$
- $U_n \subset U_{n+1}$, la dimostrazione e' per induzione su n
- $U = \cup_n U_n$, e si dimostra che $U = U_0 + \mathcal{P}_{fin}(U)$ (U e' il piu' piccolo insieme che soddisfa tale proprieta').

Un'altro problema e' come estendere un ordine lineare su U_0 ad un ordine lineare su U, anche questo puo' essere definito in modo induttivo, sulla falsariga della definizione di U_n. L'idea generale e' di definire modello canonico A, l'implementazione B e relazione di simulazione R adattando la tecnica divide-et-impera ai tipi di dato, ovvero si sta' definendo un tipo di dato in modo ricorsivo. Per l'implementazione B (mediante liste senza ripetizione) definiremo l'insieme V per interpretare *elem* per analogia con la definizione di U

- $V_{n+1} = U_0 + V_n^*$
- $V_n \subset V_{n+1}$, la dimostrazione e' per induzione su n
- $V = \cup_n V_n$, e si dimostra che $V = U_0 + V^*$ (V e' il piu' piccolo insieme che soddisfa tale proprieta').

Intutivamente la relazione di simulazione $R \subseteq V \times U$ dovrebbe soddisfare le seguenti proprieta'

- se $v \in U_0$, allora $vRu \stackrel{\Delta}{\iff} v = u$
- se $v = (v_i | i \in m) \in V^*$, allora $vRu \stackrel{\Delta}{\iff} \forall i \in m. \exists u_i \in U. v_i R u_i \wedge u = \{u_i | i \in m\} \wedge$ "v e' senza ripetizioni"

il punto delicato e' cosa vuol dire "v e' senza ripetizioni", o piu' precisamente cosa deve essere l'interpretazione $=^B: V \times V \to bool$. La risposta giusta e' che $=^B$ deve implementare $=^A$, quindi $v_i =^B v_j$ sse $u_i = u_j$. Quindi quello che si vuole richiedere e' che la lista $(u_i | i \in m)$ e' senza ripetizioni. La definizione di $R \subseteq V \times U$ con le proprieta' richiesta e' $R = \cup R_n$, dove $R_n \subseteq V_n \times U_n$ e' definita per induzione

- $vR_0u \stackrel{\Delta}{\iff} v = u$ (sia v che u sono in U_0)
- $vR_{n+1}u \stackrel{\Delta}{\iff} v = u$, quando $v \in U_0$, altrimenti
  $(v_i | i \in m) R_{n+1} u \stackrel{\Delta}{\iff} \forall i \in m. \exists u_i \in U. v_i R_n u_i \wedge u = \{u_i | i \in m\} \wedge (u_i | i \in m)$ e' senza ripetizioni.
- $R_n = R_{n+1} \cap V_n \times U_n$, la dimostrazione e' per induzione su n

Il seguente codice definisce l'implementazione della uguaglianza su V in un linguaggio ML-like

```
type U0 0 ... ; (* tipo degli elementi che non sono insiemi *)
datatype V = elem of U0 | set of V list;
- fun forall nil = true | forall (b::l) = b andalso (forall l);
val forall = fn : bool list -> bool
- fun never nil = true | never (false::l) = never l | never (true::l) = false;
val never = fn : bool list -> bool
fun existsunique nil = false
  | existsunique (false::l) = existsunique l
  | existsunique (true::l) = never l;
val existsunique = fn : bool list -> bool

fun eq (elem u1,elem u2) = u1=u2
  | eq (set l1,set l2) =
    forall (map (fn x => existsunique (map (fn y => eq(x,y)) l2)) l1) andalso
    forall (map (fn x => existsunique (map (fn y => eq(x,y)) l1)) l2)
  | eq (_,_) = false;
```

Una migliore implementazione e' data dalla rappresentazione mediante liste ordinate. Per fare cio' occorre estendere un ordine lineare su U_0 ad un ordine lineare su U. La scelta piu' ovvia e' data dall'ordine lessicografico

$$\{u_i|i \in m\} < \{u'_j|j \in n\} \stackrel{\Delta}{\iff} \begin{array}{l} \forall i.u_i < u_{i+1} \land \forall j.u'_j < u'_{j+1} \land \\ (m < n \land \forall i.u_i = u'_i) \lor (\exists k < m, n.u_k < u'_k \land \forall i < k.u_i = u'_i) \end{array}$$

Formalmente si dovrebbe definire $<$ in termini di opportuni ordini lineari $<_n \subseteq U_n \times U_n$. Nella implementazione mediante liste ordinate l'ordine lessicografico e' definibile nel modo seguente

```
fun lt (elem x,elem y) = x<y
  | lt (elem x,set l2) = true
  | lt (set nil,set(y::l2)) = true
  | lt (set(x::l1),set(y::l2)) = lt(x,y)
  | lt (set(x::l1),set(y::l2)) = x=y andalso lt(set l1,set l2)
  | lt (_,_) = false;
fun ok (elem x) = true
  | ok (set(x::nil)) = ok x
  | ok (set(x::y::l)) = ok x andalso lt(x,y) andalso ok(set(y::l));
```

Il predicato *ok* identifica gli elementi di V che rappresentano un (unico) elemento di u.

6.3 Implementazioni alternative

Chiusura transitiva di una matrice sparsa. Abbiamo dato un algoritmo per il calcolo della chiusura transitiva di una matrice M di dimensione $n \times n$ a coefficenti in un semi-anello A, supponendo che M fosse rappresentata mediante un array. Si vuole adattare l'algoritmo in mod che operi su matrici di dimensione $\mathsf{N} \times \mathsf{N}$ ma quasi ovunque nulle. Tali matrici non possono essere rappresentate mediante un array di dimensione prefissata, ma solo mediante liste, p.e. liste in $\mathsf{N} \times \mathsf{N} \times A$ ordinate nelle prime due componenti. Sviluppare i dettagli dell'algoritmo per tale rappresentazione.

OSSERVAZIONE: un possibile approccio consiste nel trovare il minimo n t.c. tutti i coefficenti non nulli di M si trovano nel quadrato $n \times n$, rappresentare M con un array $n \times n$, usare l'algoritmo visto a lezione, quindi convertire l'array $n \times n$ che rappresenta M^+ nella rappresentazione a liste. Tuttavia la complessita' tempo di tale algoritmo non e' maggiorabile in funzione del numero di coefficenti non nulli di M (p.e. se l'unico elemento non nullo di M e' in posizione $(10^9 \times 10^9)$ l'algoritmo lavorerebbe in tempo 10^{3*9}, mentre la chiusura transitiva si puo' calcolare in $O(1)$.

Tipo di dato set implementato con 3^+-alberi. Per l'implementazione basata sui 3^+-alberi ci siamo limitati a dare algoritmi con complessita' tempo $O(\log n)$ per le operazioni del tipo di dato dizionarip, escludendo le operazioni insiemistiche *union* e *intersect*. Si diano degli algoritmi per tali operazioni che lavorino in tempo $O(n_1 + n_2)$.

SUGGERIMENTO: l'approccio piu' semplice di procedere e' di trovare degli algoritmi di conversione per passare dalla rappresentazione a 3^+-albero a quella a lista ordinata e viceversa, che lavorino in tempo $O(n)$. La conversione piu' difficile e' da lista ordinata a 3^+-albero. Si consiglia di usare 2-nodi a tutti i livelli tranne che negli ultimi 2 livelli di nodi interni. Si osservi che i 3^+-alberi con 2 livelli di nodi interni possono avere un numero di nodi foglia che va' da un minimo di 4 ad un massimo di 9.

Varianti di algoritmi su grafi. Vari algoritmi su grafi (p.e. topological sorting, visite BFS e DFS) lavorano in tempo $O(V+E)$ quando si adotta una rappresentazione a liste di adiacenza. Voglia cambiare la rappresentazione del grafo usando una lista di liste di adiacenza (invece di un array di liste di adiacenza), in modo da non dover fissare a priori il numero di nodi del grafo, ma senza peggiorare la complessita' tempo di tali algoritmi. Sviluppare i dettagli di tale rappresentazione a lista di liste di adiacenza.

SUGGERIMENTO: l'idea chiave e' di associare ad ogni nodo un campo ausiliario per contenere informazioni temporanee, che negli algoritmi visti a lezione venivano tenute in array ausiliari (p.e. nodi visitati o in-degree). Descrivere l'implementazione in un linguaggio simile a C o Java.

Numero cammini in un DAG. Dato un DAG G si vuole calcolare per ogni nodo u quanti cammini ci sono che iniziano in un nodo radice ed arrivano ad u (r e' una radice se non ha nodi entranti). Adattare l'algoritmo del topological sorting per calcolare tali valori.

Implementazione di un foglio elettronico. Si implementi un foglio elettronico dove ogni cella puo' contenere una costante intera n o una operazione binaria *op* (che puo' essere $+$, $-$ o $*$) applicata a due celle (che possono anche essere la stessa). Le operazione che si desidera avere sono:

- $new()\colon cell$ crea una nuova cella e la inizializza a 0
- $set(x\colon cell, n\colon int)$ assegna n ad x
- $set_{op}(x, x_1, x_2\colon cell)$ assegna ad x l'espressione $x_1 op x_2$
- $def(x\colon cell)\colon bool$ dice se la cella x ha un valore (ci potrebbero essere dei cicli nel grafo delle dipendenze tra le celle)
- $value(x\colon cell)\colon int$ restituisce il valore della cella x, se definito

Ci sono due approcci al calcolo dei valori: si ricalcolano tutti ogni volta che si effettua una operazione *set* (in tal caso def e *value* hanno costo costante), oppure si ricalcola solo i valori strettamente necessari per rispondere ad una richiesta def o *value* (in tal caso le *set* hanno costo costante).

SUGGERIMENTO: si puo' mantenere per ogni cella anche la lista delle celle che fanno riferimento ad essa, oltr alla informazione sulle celle (al piu' 2) a cui essa fa' riferimento. In entrambi gli approcci si cerchi di ottenere degli algoritmi che dipendono solo dal numero di celle che vanno effettivamente ricalcolate, invece che dal numero di celle che fanno parte del foglio elettronico.