Università degli Studi di Udine

DIPARTIMENTO DI SCIENZE MATEMATICHE, INFORMATICHE E FISICHE
Corso di Laurea Triennale in Informatica

Tesi di Laurea

Implementazione di una variante a valori reali della codifica di Ackermann su insiemi ereditariamente finiti

Laureando:
Nicola Rizzo
`github.com/nrizzo`

Relatore:
Alberto Policriti

Anno Accademico 2017–2018

Indice

Elenco dei frammenti

Capitolo 1

Introduzione

L'obiettivo di questa tesi di laurea triennale è lo studio, la progettazione e l'implementazione di algoritmi e strutture dati per la risoluzione, o parte di una risoluzione, di un problema su insiemi, grafi e codifiche composto da due parti principali, le cui possibili applicazioni sono nella bisimulazione e nella compressione di grafi.

La prima parte del problema riguarda teoria degli insiemi e teoria dei grafi. Viene definito un universo di oggetti insiemistici a partire dall'insieme vuoto e dall'iterazione del calcolo dell'insieme delle parti, che chiamiamo insiemi ereditariamente finiti [1, p. 64-66]. Gli insiemi di questo universo hanno la proprietà di corrispondere a qualunque grafo orientato aciclico e viene dichiarata la funzione $\mathbb{N}_A$ dagli insiemi ereditariamente finiti ai numeri naturali, chiamata codifica di Ackermann, la cui efficienza e potenza è discutibile ma che rende evidenti delle proprietà interessanti degli insiemi (come viene mostrato nel capitolo 2 questa funzione è strettamente legata al *BIT predicate*, una semplice funzione sui naturali a valori 0 o 1, espressa ad esempio in [2, p. 95]).

La seconda parte invece tratta una variante di questa codifica, chiamata $\mathbb{R}_A$ (introdotta in [1, p. 197]), che al posto di avere valori interi non negativi ha valori reali, talvolta algebrici, talvolta trascendenti, in quest'ultimo caso non rappresentabili interamente da un calcolatore.[1] Ci si pone quindi un problema di approssimazione numerica, nel tentativo di calcolare questi valori con accuratezza arbitraria.

Per scrivere una libreria di aritmetica di numeri razionali a precisione arbitraria, per semplicità è utile basarsi su una più semplice libreria di interi non negativi a grandezza arbitraria: infatti la sezione 3.1 tratta dell'implementa-

[1] A differenza della codifica di Ackermann, il dominio di $\mathbb{R}_A$ potrebbe essere esteso ad un insieme più grande di insiemi, che corrispondono a dei particolari grafi orientati ciclici, ma ciò non rientra nell'ambito di studio di questa tesi.

zione di `bignat`, rappresentanti numeri naturali grandi a piacere, mentre la sezione 3.2 parla dei `bigfloat`, numeri razionali precisi a piacere. I `bignat` sono meno limitanti degli interi di macchina, quindi vengono utilizzati anche per i codici di Ackermann della prima parte del problema.

Segue una breve descrizione delle funzioni delle diverse parti di questa tesi:

- questo capitolo descrive i problemi da risolvere e la struttura del progetto;
- il capitolo 2 si prefissa di dare una base teorica al disegno delle soluzioni algoritmiche proposte;
- il capitolo 3 definisce le strutture dati utilizzate, tra cui le librerie di aritmetica, e le principali operazioni fornite da queste;
- il capitolo 4 si occupa di trasformare un codice di Ackermann nell'albero che identifica il suo insieme ereditariamente finito corrispondente;
- il capitolo 5 studia un possibile modo per approssimare $\mathbb{R}_A$;
- le note conclusive sono contenute nel capitolo 6.

Il linguaggio scelto per implementare tutte le idee è il C, e tutto il codice descritto è reperibile su `https://github.com/nrizzo/hfs-codes`.

1.1 Descrizione del problema

Definizione 1.1.1 (Chiusura transitiva)**.** Dato un insieme x, definiamo ricorsivamente la sua chiusura transitiva come

$$\mathrm{trCl}(x) =_{\mathrm{Def}} x \cup \bigcup_{y \in x} \mathrm{trCl}(y).$$

Definizione 1.1.2 (Insieme ereditariamente finito)**.** Diciamo che l'insieme x è ereditariamente finito se, per definizione, la sua chiusura transitiva è finita.

L'universo di insiemi ereditariamente finiti che maneggeremo è costruito a partire dall'insieme vuoto e dalla procedura di insieme delle parti $\mathcal{P}$.

Definizione 1.1.3 (HF)**.** Definiamo l'insieme HF come

$$\mathsf{HF} =_{\mathrm{Def}} \bigcup_{n \in \mathbb{N}} \mathsf{HF}_n,$$

i cui livelli HF_n sono definiti induttivamente come segue:

$$\begin{aligned} \mathsf{HF}_0 &=_{\text{Def}} \emptyset, \\ \mathsf{HF}_{m+1} &=_{\text{Def}} \mathcal{P}\big(\mathsf{HF}_m\big) \quad \forall m \in \mathbb{N}. \end{aligned}$$

Definizione 1.1.4 (Codifica di Ackermann)**.** Per ogni $x \in \mathsf{HF}$, il numero di Ackermann di x (o codice di x) è definito ricorsivamente come

$$\mathbb{N}_A(x) =_{\text{Def}} \sum_{y \in x} 2^{\mathbb{N}_A(y)}.$$

Nel prossimo capitolo dimostreremo che tra le proprietà interessanti di $\mathbb{N}_A$ c'è quella di essere una biezione nei confronti dei numeri naturali: da ora in poi indichiamo con h_n l'insieme di HF tale che $\mathbb{N}_A(h_n) = n$.

Definizione 1.1.5 (Grafo di appartenenza)**.** Il grafo di appartenenza di un insieme $x \in \mathsf{HF}$ è il grafo orientato $G = (V, E)$ tale che

$$V = \mathrm{trCl}(x) \qquad \text{e} \qquad E = \big\{\langle u, v\rangle \colon u, v \in V \mid u \ni v\big\}.$$

Problema 1. Dato n, si costruisca il grafo di appartenenza di h_n.

Definizione 1.1.6 ($\mathbb{R}_A$)**.** Per ogni $x \in \mathsf{HF}$, definiamo una variante della codifica di Ackermann come

$$\mathbb{R}_A(x) =_{\text{Def}} \sum_{y \in x} 2^{-\mathbb{R}_A(y)}.$$

Problema 2. Dato $n \in \mathbb{N}$, avendo il risolto il problema 1 per n, si calcoli un'approssimazione di $\mathbb{R}_A(h_n)$ precisa a piacere.

Specifiche di input e di output I caratteri dell'espansione binaria di n vengono letti da *standard input*, mentre viene stampato su *standard output* sia il grafo di appartenenza di h_n in formato DOT che l'approssimazione di $\mathbb{R}_A(h_n)$ come un intervallo di numeri in cui si trova sicuramente la codifica esatta (per ragioni implementative gli estremi sono espressi come mantissa in notazione esadecimale, moltiplicata per una potenza di 2^{32}).

1.2 Il progetto

Il progetto è reperibile su Github, all'indirizzo `https://github.com/nrizzo/hfs-codes`. Avendo installato `git` sulla propria macchina, il seguente comando da terminale clona la repository del progetto.

```
$ git clone https://github.com/nrizzo/hfs-codes
```

1.2.1 Come compilare

Il progetto utilizza un `Makefile` per produrre l'algoritmo risolutivo (chiamato `algoritmo_risolutivo`). I seguenti comandi, da eseguire nella cartella del progetto, generano l'eseguibile ed eliminano i file temporanei.

```
$ make
$ make clean
```

1.2.2 Struttura

- `src/main.c` contiene l'algoritmo risolutivo e la procedura per leggere l'input n e trovare i codici dei figli di h_n;
- i file in `src/bignat` implementano numeri non negativi di grandezza arbitraria;
- i file in `src/bigfloat` implementano numeri razionali di precisione arbitraria;
- i file in `src/interval` implementano intervalli di razionali e le operazioni di somma di intervalli e di approssimazione del reciproco dell'elevamento a potenza di 2;
- i file in `src/list` implementano le liste;
- i file in `src/memgraph`, assieme a quelli in `src/memgraph/memnode` implementano il grafo di appartenenza.

1.2.3 Algoritmo risolutivo

L'algoritmo risolutivo, mostrato nel frammento 1.1, legge le cifre binarie di n da *standard input*, genera il grafo di appartenenza di h_n e calcola $\mathbb{R}_A(h_n)$.

```
0  int main()
1  {
2          struct dl_list_b *list; /* lista dei figli di h_n */
3          struct memgraph *G; /* grafo di appartenenza di h_n */
4
5          list = dllb_scanchildren();
6
7          G = memgraph_create(list);
8          memgraph_build(G);
9          memgraph_calcrack(G, DE, DN, DA);
10
11         memgraph_printDOT(G);
12         interval_print(G->rcode); printf("\n");
13
14         dllb_destroy(list);
15         memgraph_destroy(G);
16 }
```

Frammento 1.1: `main`.

Capitolo 2

Considerazioni di base

Questo capitolo si concentra sul trovare e dimostrare proprietà utili di insiemi ereditariamente finiti e delle codifiche proposte, in modo da fornire una base teorica alla correttezza e allo studio della complessità degli algoritmi risolutivi proposti nei capitoli successivi.

2.1 Insiemi ereditariamente finiti

Teorema 2.1.1. $\mathsf{HF}_n \subset \mathsf{HF}_{n+1}$ per ogni $n \in \mathbb{N}$.

Dimostrazione. La dimostrazione che $\mathsf{HF}_n \subseteq \mathsf{HF}_{n+1}$ per ogni $n \in \mathbb{N}$ procede per induzione su n.

caso base: se $n = 0$, $\mathsf{HF}_0 = \emptyset$ e $\emptyset \subseteq A$ per qualunque insieme A, anche HF_1;

ipotesi induttiva: sia valida la tesi per $n = m$, cioè $\mathsf{HF}_m \subseteq \mathsf{HF}_{m+1}$;

passo induttivo: per la definizione 1.1.3 $\mathsf{HF}_{m+1} = \mathcal{P}(\mathsf{HF}_m)$, cioè l'elemento generico x di HF_{m+1} è

$$\begin{aligned} x = \{x_1, \ldots, x_k\} \qquad & x_1, \ldots, x_k \in \mathsf{HF}_m \\ & x_1 \neq \cdots \neq x_k \\ & k \in \{0, \ldots, |\mathsf{HF}_m|\}, \end{aligned}$$

ma per ipotesi induttiva $x_1, \ldots x_k \in \mathsf{HF}_{m+1}$ e, poiché HF_{m+2} è definito come l'insieme delle parti di HF_{m+1}, $\{x_1, \ldots, x_k\}$ è un sottoinsieme di HF_{m+1} e $x \in \mathsf{HF}_{m+2}$.

Per concludere, è facile notare che $A \neq \mathcal{P}(A)$ per qualunque insieme finito A (infatti $|\mathcal{P}(A)| = 2^{|A|}$), cioè $\mathsf{HF}_n \neq \mathsf{HF}_{n+1}$ per ogni $n \in \mathbb{N}$. □

Quindi la serie $\{\mathsf{HF}_n, n \in \mathbb{N}\}$ consiste di insiemi ereditariamente finiti di cardinalità crescente (per iterazione dell'elevamento a potenza di due) e converge a HF. Una funzione utile per indicare la cardinalità di HF_n è la seguente $\beth$: $\mathbb{N} \to \mathbb{N}$ (tetrazione di 2):

$$\beth(k) =_{\text{Def}} \begin{cases} 0 & \text{se } k = 0, \\ 2^{\beth(k-1)} & \text{altrimenti.} \end{cases}$$

Definizione 2.1.2 (Rango)**.** Definiamo il rango di un insieme $x \in \mathsf{HF}$, che indichiamo con $\text{rank}(x)$, come il più piccolo $n \in \mathbb{N}$ tale che $x \in \mathsf{HF}_{n+1}$. Alternativamente, è l'intero n per cui vale $x \in \mathsf{HF}_{n+1} \setminus \mathsf{HF}_n$.

Lemma 2.1.3. Dato $x \in \mathsf{HF}$, se $x \ni y$ allora $\text{rank}(x) > \text{rank}(y)$.

Dimostrazione. Per un generico $z \in \mathsf{HF}$, se $z \in \mathsf{HF}_n$ allora $\text{rank}(z) < n$ perché:

- $z \in \mathsf{HF}_i$ per ogni intero $i > \text{rank}(z)$, per il teorema 2.1.1 e per la definizione di rango;
- $z \notin \mathsf{HF}_j$ per ogni intero $j \leq \text{rank}(z)$, verificabile banalmente dalla definizione di rango;

$x \in \mathsf{HF}_{\text{rank}(x)+1}$, cioè x è un sottoinsieme di $\mathsf{HF}_{\text{rank}(x)}$ della forma

$$\begin{aligned} x = \{x_1, \ldots, x_k\}, \qquad & x_1, \ldots, x_k \in \mathsf{HF}_{\text{rank}(x)} \\ & x_1 \neq \cdots \neq x_k \\ & k \in \{0, \ldots, |\mathsf{HF}_{\text{rank}(x)}|\}; \end{aligned}$$

se $y \in x$ allora è un elemento di $\mathsf{HF}_{\text{rank}(x)}$ e perciò $\text{rank}(y) < \text{rank}(x)$. □

Lemma 2.1.4. Dato $x \in \mathsf{HF}$, se $\text{rank}(x) \geq 1$ allora x contiene almeno un elemento di rango $\text{rank}(x) - 1$.

Dimostrazione. Dato $x \in \mathsf{HF}$, x non contiene alcun elemento se e solo se è l'insieme vuoto, unico elemento di rango 0. Altrimenti, per la definizione alternativa 2.1.2 di rango e per la definizione 1.1.3, $x \in \mathsf{HF}_{\text{rank}(x)+1} \setminus \mathsf{HF}_{\text{rank}(x)}$ e x è un sottoinsieme di $\mathsf{HF}_{\text{rank}(x)}$ ma non di $\mathsf{HF}_{\text{rank}(x)-1}$, quindi

$$\begin{aligned} x \subseteq \mathsf{HF}_{\text{rank}(x)} \quad &\Longrightarrow \quad \forall y \in x \quad y \in \mathsf{HF}_{\text{rank}(x)}, \\ x \nsubseteq \mathsf{HF}_{\text{rank}(x)-1} \quad &\Longrightarrow \quad \exists y \in x \quad y \notin \mathsf{HF}_{\text{rank}(x)-1}, \end{aligned}$$

cioè esiste un $y \in x$ tale che $y \in \mathsf{HF}_{\text{rank}(x)} \setminus \mathsf{HF}_{\text{rank}(x)-1}$, cioè ha rango $\text{rank}(x) - 1$. □

Lemma 2.1.5. Dato $x \in \mathsf{HF}$, la funzione $r\colon \mathsf{HF} \to \mathbb{N}$ così definita:

$$r(x) = \begin{cases} 0 & \text{se } x = \emptyset, \\ \max\limits_{y \in x}\big(r(y) + 1\big) & \text{altrimenti,} \end{cases}$$

equivale al rango di x.

Dimostrazione. Se $x \in \mathsf{HF}$ è l'insieme vuoto allora ha rango 0, perché è l'unico elemento di HF_1; altrimenti, per i lemmi 2.1.3 e 2.1.4 nessun elemento di x può avere rango uguale o superiore a $\operatorname{rank}(x)$ ed esiste sicuramente un elemento di rango $\operatorname{rank}(x) - 1$, il massimo possibile, rendendo banalmente vera la formula. □

Lemma 2.1.6. Dato $x \in \mathsf{HF}_n$, $\operatorname{trCl}(x) \in \mathsf{HF}_n$ per ogni $n \in \mathbb{N}$.

Dimostrazione. La dimostrazione procede per induzione su n.

caso base: se $n = 0$, x è l'insieme vuoto e la sua chiusura transitiva è anch'essa l'insieme vuoto;

ipotesi induttiva: sia valida la tesi per $n = m$;

passo induttivo: dato $y \in \mathsf{HF}_{m+1}$, per la definizione 1.1.3 di HF_{m+1} e per la definizione di insieme delle parti questo è un sottoinsieme qualsiasi di HF_m della forma

$$\begin{aligned} y = \{y_1, \ldots, y_k\}, \qquad & y_1, \ldots, y_k \in \mathsf{HF}_m \\ & y_1 \neq \cdots \neq y_k \\ & k \in \{0, \ldots, |\mathsf{HF}_m|\}, \end{aligned}$$

quindi la chiusura transitiva di y si può riscrivere come

$$\operatorname{trCl}(y) = \{y_1, \ldots, y_k\} \cup \bigcup_{i=i}^{k} \operatorname{trCl}(y_i);$$

ma $y_1, \ldots, y_k$ sono elementi di HF_m e $\operatorname{trCl}(y_1), \ldots, \operatorname{trCl}(y_k)$ sono elementi di HF_m per ipotesi induttiva, cioè la loro unione contiene elementi di HF_{m-1}, che sono anche elementi di HF_m per il teorema 2.1.1.

□

Una conseguenza banale di questo lemma è che per ogni elemento di HF la sua chiusura transitiva è a sua volta un insieme ereditariamente finito.

Teorema 2.1.7. Dato $x \in \mathsf{HF}$, la chiusura transitiva di x ha il suo stesso rango, in simboli $\operatorname{rank}\big(\operatorname{trCl}(x)\big) = \operatorname{rank}(x)$.

Dimostrazione. Ha senso chiedersi quale sia il rango di $\operatorname{trCl}(x)$ perché, per il lemma 2.1.6, esso appartiene ad $\mathsf{HF}_{\operatorname{rank}(x)+1}$. Se $x = \emptyset$, allora $\operatorname{trCl}(x) = \emptyset$ e il rango dell'insieme vuoto è 0, altrimenti x contiene almeno un elemento e il suo rango è maggiore o uguale a 1. Per il lemma 2.1.4 x contiene almeno un elemento di rango $\operatorname{rank}(x) - 1$ e per la definizione di trCl ogni elemento di x è contenuto anche dalla chiusura transitiva: cioè, per il lemma 2.1.3,

$$\operatorname{rank}(\operatorname{trCl}(x)) \geq \operatorname{rank}(x).$$

Inoltre per la definizione di rango

$$\operatorname{trCl}(x) \in \mathsf{HF}_{\operatorname{rank}(x)+1} \implies \operatorname{rank}(\operatorname{trCl}(x)) \leq \operatorname{rank}(x),$$

perciò $\operatorname{rank}(\operatorname{trCl}(x)) = \operatorname{rank}(x)$. □

2.2 Codifica di Ackermann

Lemma 2.2.1 (di iniettività)**.** Per ogni $n \in \mathbb{N}$, vale che

$$\forall x, y \in \mathsf{HF}_n \qquad x \neq y \implies \mathbb{N}_A(x) \neq \mathbb{N}_A(y).$$

Dimostrazione. La dimostrazione procede per induzione su n.

casi base: se $n = 0$, HF_0 non ha nessun elemento, rendendo vera la tesi; per scrupolo, se $n = 1$ allora HF_1 contiene solo un elemento, altra condizione sufficiente per verificare la tesi;

ipotesi induttiva: sia valida la tesi per $n = m$, cioè

$$\forall x, y \in \mathsf{HF}_m \qquad x \neq y \implies \mathbb{N}_A(x) \neq \mathbb{N}_A(y);$$

passo induttivo: siano x e y due generici elementi di HF_{m+1}, che per la definizione 1.1.3 contiene tutti i possibili sottoinsiemi di HF_m, perché $\mathsf{HF}_{m+1} = \mathcal{P}(\mathsf{HF}_m)$, quindi x e y sono della forma

$$\begin{aligned} x = \{x_1, \ldots, x_j\} \qquad & x_1, \ldots, x_j \in \mathsf{HF}_m, \\ & x_1 \neq \cdots \neq x_j, \\ & j \in \{0, \ldots, |\mathsf{HF}_m|\} \text{ e} \\ y = \{y_1, \ldots, y_k\} \qquad & y_1, \ldots, y_k \in \mathsf{HF}_m, \\ & y_1 \neq \cdots \neq y_k, \\ & k \in \{0, \ldots, |\mathsf{HF}_m|\}, \end{aligned}$$

e su $x_1, \ldots, x_j$ e su $y_1, \ldots, y_k$ si può applicare l'ipotesi induttiva, cioè $\mathbb{N}_A(x_1) \neq \cdots \neq \mathbb{N}_A(x_j)$ e $\mathbb{N}_A(y_1) \neq \cdots \neq \mathbb{N}_A(y_k)$; se $x \neq y$, allora esiste $z \in \mathsf{HF}_m$ tale che $z \in x$ e $z \notin y$ oppure $z \notin x$ e $z \in y$, ma in entrambi i casi, per la forma di $\mathbb{N}_A$, ciò implica che $\mathbb{N}_A(x)$ e $\mathbb{N}_A(y)$ differiscono nella $(\mathbb{N}_A(z)+1)$-esima cifra della loro espansione binaria, cioè $\mathbb{N}_A(x) \neq \mathbb{N}_A(y)$.

□

Lemma 2.2.2 (di suriettività)**.** Per ogni $n \in \mathbb{N}$ tale che $n \geq 1$, vale che

$$\forall i \in \{0, 1, \ldots, \beth(n) - 1\} \qquad \big(\exists x \in \mathsf{HF}_n \quad \mathbb{N}_A(x) = i\big).$$

Dimostrazione. La dimostrazione procede per induzione su n.

caso base: se $n = 1$, l'insieme $\{0, 1, \ldots, \beth(n) - 1\}$ contiene solamente lo 0, HF_1 contiene solamente l'insieme vuoto e il codice di Ackermann dell'insieme vuoto è lo 0;

ipotesi induttiva: sia vera la tesi per $n = m$;

passo induttivo: sia x l'elemento generico di HF_{m+1}, allora per la definizione 1.1.3 $\mathsf{HF}_{m+1} = \mathcal{P}(\mathsf{HF}_m)$ e x è un sottoinsieme qualunque di HF_m della forma

$$\begin{aligned} x = \{x_1, \ldots, x_k\}, \qquad & x_1, \ldots, x_k \in \mathsf{HF}_m \\ & x_1 \neq \cdots \neq x_k \\ & k \in \{0, \ldots, |\mathsf{HF}_m|\}, \end{aligned}$$

per il lemma 2.2.1 di iniettività $\mathbb{N}_A(x_1) \neq \cdots \neq \mathbb{N}_A(x_k)$ e per ipotesi induttiva per ogni intero da 0 a $\beth(m)-1$ esiste un elemento di HF_m con tale codice, cioè $\mathbb{N}_A$ è una biezione da HF_m a $\{0, \ldots, \beth(m) - 1\}$; infine $\mathbb{N}_A(x)$ è una somma di potenze di 2 con esponenti distinti e qualunque tra 0 e $\beth(m)-1$, cioè corrisponde all'espansione binaria di un qualunque intero tra 0 e $\beth(m+1) - 1$, verificando la tesi per $n = m + 1$.

□

Teorema 2.2.3. La funzione $\mathbb{N}_A \colon \mathsf{HF} \to \mathbb{N}$ è una biezione.

Dimostrazione. L'iniettività e la suriettività sono facilmente verificabili per il teorema 2.1.1 e i lemmi 2.2.1 e 2.2.2. □

Teorema 2.2.4. Dato $x \in \mathsf{HF}$,

$$\mathbb{N}_A(x) \in \Big\{\beth\big(\operatorname{rank}(x)\big), \ldots, \beth\big(\operatorname{rank}(x) + 1\big) - 1\Big\}.$$

Dimostrazione. Per i lemmi 2.2.1 e 2.2.2, per ogni $n \in \mathbb{N}$ la codifica di Ackermann è una biezione da HF_n a $\{0, \ldots, \beth(n-1)-1\}$. Inoltre per ogni $x \in \mathsf{HF}$ vale che $x \in \mathsf{HF}_{\mathrm{rank}(x)+1}$, cioè il suo codice è compreso tra 0 e $\beth\big(\mathrm{rank}(x)+1\big)-1$, e $x \notin \mathsf{HF}_{\mathrm{rank}(x)}$, che esclude i codici tra 0 e $\beth\big(\mathrm{rank}(x)\big)-1$. □

Lemma 2.2.5. Dati $x, y \in \mathsf{HF}$ tali che $y \in x$, vale che

$$\mathbb{N}_A(y) \leq \log_2\big(\mathbb{N}_A(x)\big).$$

Dimostrazione. È sufficiente notare che se $y \in x$ allora x ha almeno un elemento, e la formula per il calcolo di $\mathbb{N}_A(x)$ si può riscrivere come

$$\mathbb{N}_A(x) = \left(\sum_{z \in x, z \neq y} 2^{\mathbb{N}_A(z)}\right) + 2^{\mathbb{N}_A(y)},$$

per cui vale

$$\mathbb{N}_A(x) \geq 2^{\mathbb{N}_A(y)} \quad \iff \quad \mathbb{N}_A(y) \leq \log_2\big(\mathbb{N}_A(x)\big).$$

□

Data la forma di $\mathbb{N}_A$ e la sua biettività si possono trarre proprietà interessanti, poiché il codice di un insieme ereditariamente finito è strettamente legato alla sua rappresentazione binaria:

- l'insieme x contiene y se e solo se il $(\mathbb{N}_A(y)+1)$-esimo bit meno significativo di $\mathbb{N}_A(x)$ è 1, essendo $\mathbb{N}_A(x)$ una somma di potenze di due con tutti gli esponenti diversi tra loro e corrispondenti ad uno specifico elemento di x;
- la cardinalità dell'insieme x corrisponde al numero di 1 che occorrono nella rappresentazione binaria di $\mathbb{N}_A(x)$;
- la posizione della cifra di valore 1 più significativa in notazione binaria, quindi anche il minor numero di cifre necessarie a rappresentare il codice, dipendono dal rango dell'insieme; infatti, dato $\mathbb{N}_A(x)$, la funzione $\beth\colon \mathbb{N} \to \mathbb{N}$ definita come

  $$\beth(n) =_{\mathrm{Def}} \begin{cases} 0 & \text{se } n = 0 \\ \beth\big(\lfloor \log_2 n \rfloor\big) + 1 & \text{altrimenti,} \end{cases}$$

  restituisce il rango di x; ancora più specificatamente, il numero di cifre di $\mathbb{N}_A(x)$ dipende dal massimo codice di Ackermann dei figli di x, cioè è $\max_{y \in x}(\mathbb{N}_A(y)+1)$, o alternativamente $\lfloor \log_2 \mathbb{N}_A(x) \rfloor + 1$ se $\mathbb{N}_A(x) > 0$.

Queste considerazioni rendono utile, specialmente da parte di un calcolatore, la rappresentazione degli insiemi ereditariamente finiti come i loro rispettivi codici di Ackermann.

2.3 Grafo di appartenenza

Dato $x \in \mathsf{HF}$, è importante notare che x non è un vertice del suo grafo di appartenenza, ma che tutti gli elementi di x (e gli elementi degli elementi, ricorsivamente) sono presenti grazie alla chiusura transitiva. Inoltre, più insiemi possono avere lo stesso grafo di appartenenza.

Teorema 2.3.1. Dato $x \in \mathsf{HF}$, il suo grafo di appartenenza è aciclico.

Dimostrazione. Se, per assurdo, esistesse un ciclo, questo implicherebbe una catena di appartenenze della forma $y_1 \ni y_2 \ni \cdots \ni y_k \ni y_1$, ma per il lemma 2.1.3 ciò equivale a $\operatorname{rank}(y_1) < \operatorname{rank}(y_2) < \cdots < \operatorname{rank}(y_k) < \operatorname{rank}(y_1)$, che è assurdo. □

Lemma 2.3.2. Per ogni $n \in \mathbb{N}$, dato $x \in \mathsf{HF}_n$ la chiusura transitiva di ogni elemento di $\operatorname{trCl}(x)$ è un sottoinsieme di $\operatorname{trCl}(x)$, in simboli

$$\forall y \in \operatorname{trCl}(x) \qquad \operatorname{trCl}(y) \subseteq \operatorname{trCl}(x).$$

Dimostrazione. La dimostrazione procede per induzione su n.

casi base: se $n = 0$, HF_0 è vuoto; se $n = 1$, HF_1 contiene l'insieme vuoto, e la sua chiusura transitiva è lo stesso insieme vuoto, sottoinsieme di qualunque altro insieme;

ipotesi induttiva: sia vera la tesi per $n \leq m$;

passo induttivo: sia $x \in \mathsf{HF}_{m+1}$, allora per la definizione 1.1.3 $\mathsf{HF}_{m+1} = \mathcal{P}(\mathsf{HF}_m)$ e x è un sottoinsieme qualunque di HF_m della forma

$$\begin{aligned} x = \{x_1, \ldots, x_k\}, \qquad & x_1, \ldots, x_k \in \mathsf{HF}_m \\ & x_1 \neq \cdots \neq x_k \\ & k \in \{0, \ldots, |\mathsf{HF}_m|\}, \end{aligned}$$

e per la definizione di chiusura transitiva 1.1.1

$$\operatorname{trCl}(x) = \{x_1, \ldots, x_k\} \cup \bigcup_{i=1}^{k} \operatorname{trCl}(x_i);$$

ogni elemento di $\operatorname{trCl}(x)$ è uno tra $x_1, \ldots x_k$, e la sua chiusura transitiva è un sottoinsieme di $\operatorname{trCl}(x)$ per definizione, oppure è un elemento di una tra le chiusure transitive $\operatorname{trCl}(x_1), \ldots \operatorname{trCl}(x_k)$, su cui si può applicare l'ipotesi induttiva.

□

Teorema 2.3.3. Dato $x \in \mathsf{HF}$, due vertici del grafo di appartenenza di x sono distinti se e solo se i due insiemi dei rispettivi figli sono distinti.

Dimostrazione. ($\Longrightarrow$) Dati due vertici distinti $y_1, y_2 \in V$, definiamo gli insiemi dei loro figli nel grafo come $N_{y_1} = \{z \in V \mid \langle y_1, z\rangle \in E\}$ e $N_{y_2} = \{z \in V \mid \langle y_2, z\rangle \in E\}$; vale che

$$y_1 \neq y_2 \qquad \Longrightarrow \qquad \exists z \in y_1 \; z \notin y_2 \quad \vee \quad \exists z \in y_2 \; z \notin y_1.$$

Nel caso in cui $z \in y_1$ e $z \notin y_2$, per la definizione di grafo di appartenenza e per il lemma 2.3.2 riguardante la chiusura transitiva, $z \in V$ e

$$\begin{aligned} z \in y_1 \quad &\Longrightarrow \quad \langle y_1, z\rangle \in E \quad \Longrightarrow \quad z \in N_{y_1}, \\ z \notin y_2 \quad &\Longrightarrow \quad \langle y_2, z\rangle \notin E \quad \Longrightarrow \quad z \notin N_{y_2}; \end{aligned}$$

quindi $N_{y_1} \neq N_{y_2}$. Il caso in cui $z \notin y_1$ e $z \in y_2$ è simmetrico.

($\Longleftarrow$) Dati $y_1, y_2 \in V$ tali che $N_{y_1} \neq N_{y_2}$, vale che

$$\exists z \in V \quad \big(\langle y_1, z\rangle \in E \wedge \langle y_2, z\rangle \notin E\big) \quad \vee \quad \big(\langle y_1, z\rangle \notin E \wedge \langle y_2, z\rangle \in E\big).$$

Nel primo caso, per la definizione di grafo di appartenenza, $z \in y_1$ e $z \notin y_2$, cioè $y_1 \neq y_2$; nel secondo caso $z \notin y_1$ e $z \in y_2$, cioè $y_1 \neq y_2$. □

Teorema 2.3.4. Dato $x \in \mathsf{HF}$, da ogni nodo del suo grafo di appartenenza si può raggiungere l'insieme vuoto, che è l'unico pozzo.

Dimostrazione. Se $x = \emptyset$ allora l'unico vertice del grafo di appartenenza è l'insieme vuoto; altrimenti $\operatorname{rank}(x) \geq 1$ e per il lemma 2.1.4 è facile dimostrare (per induzione) che x ha almeno un figlio di rango $\operatorname{rank}(x) - 1$, che ha a sua volta un figlio di rango $\operatorname{rank}(x) - 2$, fino ad arrivare a 0: l'unico elemento di rango 0 è proprio l'insieme vuoto. Quest'ultimo è l'unico pozzo, perché è l'unico vertice senza figli per il teorema 2.3.3. □

Teorema 2.3.5. Dato $x \in \mathsf{HF}$, l'altezza nel suo grafo di appartenenza $G = (V, E)$ di un vertice corrisponde al rango del vertice stesso.

Dimostrazione. La dimostrazione procede per induzione sul rango del vertice generico $v \in V$.

caso base: se $\operatorname{rank}(v) = 0$ allora $v = \emptyset$, v non contiene alcun figlio e la sua altezza è 0;

ipotesi induttiva: supponiamo la tesi sia valida per tutti gli elementi di rango inferiore o uguale a m;

passo induttivo: se $\text{rank}(v) = m + 1$, qualunque figlio di v, sicuramente anch'esso un vertice del grafo per la definizione di chiusura transitiva, deve avere rango minore o uguale a m per il lemma 2.1.3 su cui si può applicare l'ipotesi induttiva; per lemma 2.1.4 v ha almeno un figlio di rango massimo $\text{rank}(v) = m$, cioè di altezza m, quindi v ha altezza $m + 1$.

□

Lemma 2.3.6. Dato $x \in \mathsf{HF}$,

$$|\text{trCl}(x)| \geq \text{rank}(x).$$

Dimostrazione. La dimostrazione procede per induzione sul rango di x.

caso base: se $\text{rank}(x) = 0$, allora $x = \emptyset$, unico elemento di HF_1, $\text{trCl}(\emptyset) = \emptyset$ e $|\text{trCl}(x)| = \text{rank}(x) = 0$;

ipotesi induttiva: la tesi è vera per tutti gli $x \in \mathsf{HF}$ tali che $\text{rank}(x) = m$;

passo induttivo: se $\text{rank}(x) = m + 1$ allora, per il lemma 2.1.3, x contiene y tale che $\text{rank}(y) = m$ e, per la definizione di chiusura transitiva,

$$x \cup \text{trCl}(y) \subseteq \text{trCl}(x);$$

su y si può applicare l'ipotesi induttiva, cioè $|\text{trCl}(y)| \geq m$, e l'insieme $x \setminus \text{trCl}(y)$ contiene almeno un elemento, perché $y \in x$ e $y \notin \text{trCl}(y)$, quindi

$$|x \cup \text{trCl}(y)| \geq m + 1 \quad \Longrightarrow \quad |\text{trCl}(x)| \geq \text{rank}(x).$$

□

Lemma 2.3.7. Dato $x \in \mathsf{HF}$,

$$|\text{trCl}(x)| \leq \max_{y \in V} \mathbb{N}_A(y) + 1.$$

Dimostrazione. È facile notare come ogni insieme appartenente a $\text{trCl}(x)$ è un discendente di x ed il suo codice è minore o uguale del codice massimo dei figli di x: per la biettività di $\mathbb{N}_A$ dimostrata dal teorema 2.2.3, il numero massimo di elementi di $\text{trCl}(x)$ è $\max_{y \in V}(\mathbb{N}_A(y) + 1)$. □

Lemma 2.3.8. Dati $x \in \mathsf{HF}$ e il suo grafo di appartenenza $G = (V, E)$,

$$|E| \geq |V| - 1.$$

Dimostrazione. Per il lemma 2.1.3, $\mathrm{trCl}(x)$ ha lo stesso rango di x e, per il lemma 2.1.4, x contiene almeno un elemento di rango $\mathrm{rank}(x) - 1$ vertice di G: per il teorema 2.3.4 questo vertice ha altezza $\mathrm{rank}(x) - 1$, cioè esistono almeno altrettanti archi. □

Lemma 2.3.9. Dato $x \in \mathsf{HF}$ tale che $\mathrm{rank}(x) \geq 3$, dato il suo grafo di appartenenza $G = (V, E)$ e definito $k = \max_{y \in V}(\mathbb{N}_A(y) + 1)$,

$$|E| < k \log_2 k.$$

Dimostrazione. Intuitivamente, il grafo di appartenenza con più archi possibili è tale che ogni elemento con codice minore o uguale del codice massimo dei figli di x è un vertice. Al posto di chiederci la quantità dei figli di un vertice generico, è utile chiedersi il numero dei genitori: sapendo che per il lemma 2.2.5, dati $y, z \in V$ tali che $y \in z$ vale $\mathbb{N}_A(z) \geq 2^{\mathbb{N}_A(y)}$, allora i vertici che possono avere genitori nel grafo hanno il proprio codice compreso tra 0 e $\lfloor \log_2 k \rfloor$, estremi inclusi.

Approssimando per eccesso il numero dei genitori di un nodo generico a tutto $|V|$, allora

$$|E| < k \cdot \log_2 k.$$

□

Il limite superiore trovato da questo lemma può effettivamente essere raggiunto asintoticamente, poiché osservando il grafo di appartenenza di un insieme ereditariamente finito con codice $2^m - 1$ si può notare che approssimativamente i primi $\log m$ figli con codice più piccolo hanno tutti circa $m/2$ genitori.

Teorema 2.3.10. Dato $x \in \mathsf{HF}$, tale che $\mathrm{rank}(x) \geq 3$, dato il suo grafo di appartenenza $G = (V, E)$, e definito $k = \max_{y \in V}(\mathbb{N}_A(y) + 1)$,

$$\mathrm{rank}(x) \leq |V| \leq k \qquad \text{e} \qquad |V| - 1 \leq |E| \leq k \cdot \log_2 k.$$

Dimostrazione. I limiti sono dimostrati dai lemmi 2.3.6, 2.3.7, 2.3.8 e 2.3.9. □

Teorema 2.3.11. Dato $x \in \mathsf{HF}$ e il suo grafo di appartenenza $G = (V, E)$, l'ordinamento decrescente dei vertici secondo i rispettivi codici di Ackermann è topologico.

Dimostrazione. Siano $y, z \in V$ tali che $\langle y, z \rangle \in E$. Per la definizione di grafo di appartenenza $y \ni z$ e per il lemma 2.2.5

$$\mathbb{N}_A(y) \geq 2^{\mathbb{N}_A(z)} \quad \Longrightarrow \quad \mathbb{N}_A(y) > \mathbb{N}_A(z).$$

□

2.4 Variante a valori reali

Lemma 2.4.1. Dato $x \in \mathsf{HF}$,

$$\mathbb{R}_A(x) = \sum_{y \in x} \mathbb{R}_A\big(\{y\}\big).$$

Dimostrazione. È facile notare come, per la definizione 1.1.6 di $\mathbb{R}_A$, per ogni $y \in \mathsf{HF}$ vale che $\mathbb{R}_A(\{y\}) = 2^{-\mathbb{R}_A(y)}$. La tesi è una diretta conseguenza poiché

$$\mathbb{R}_A(x) = \sum_{y \in x} 2^{-\mathbb{R}_A(y)} = \sum_{y \in x} \mathbb{R}_A\big(\{y\}\big).$$

□

Capitolo 3

Strutture dati

Questo capitolo si occupa della presentazione delle strutture dati più importanti utilizzate che sono:

- interi non negativi grandi a piacere (`struct bignat`) con l'operazione di *BIT predicate*, che dato $i \in \mathbb{N}$ restituisce l'$(i+1)$-esima cifra meno significativa dell'espansione binaria; inoltre i `bignat` supportano le operazioni aritmetiche di base quali addizione, sottrazione (se il risultato è ancora non negativo), moltiplicazione e divisione per un intero tra 1 e $2^{32} - 1$;

- numeri razionali precisi a piacere (`struct bigfloat`), rappresentati come segno, mantissa ed esponente in base 2^{32}, che supportano operazioni aritmetiche semplici e di approssimazione per difetto o per eccesso ad una data cifra;

- liste doppiamente concatenate di `bignat` (`struct dl_list_b`) o di `memnode`, nodi del grafo di appartenenza (`struct dl_list_n`);

- il grafo di appartenenza (`struct memgraph`), rappresentato come una lista di nodi (`struct memnode`) che a loro volta contengono la propria lista di adiacenza (gli algoritmi per il calcolo del grafo di appartenenza sono discussi nel capitolo 4);

- intervalli, aperti o chiusi, di `bigfloat` (`struct interval`), utili ad approssimare $\mathbb{R}_A$ (gli algoritmi che li coinvolgono sono discussi nel capitolo 5).

3.1 Interi non negativi grandi a piacere: `bignat`

In questa sezione si studiano algoritmi e la loro implementazione in C per l'aritmetica di naturali grandi a piacere.[1]

3.1.1 La notazione posizionale

Per indicare la rappresentazione di un numero u in una data base b ($b \in \mathbb{N}$ e $b \geq 2$) utilizziamo la seguente notazione, chiamata posizionale per i diversi pesi che vengono associati alle cifre in base alla loro posizione:

$$(u_{n-1} \dots u_1 u_0)_b =_{\text{Def}} \sum_{i=0}^{n-1} u_i \cdot b^i, \qquad u_0, u_1, \dots, u_{n-1} \in \{0, 1, \dots, b-1\}.$$

Definizione 3.1.1 (Intero di n cifre in base b)**.** Dato u, diciamo che esso è un intero di n cifre in base b, con $b > 0$ e intero, se può essere scritto con n cifre in notazione posizionale in base b, in simboli

$$u = (u_{n-1} \dots u_1 u_0)_b.$$

Equivalentemente, ogni intero minore o uguale di b^n è un intero di n cifre in base b.

È importante notare che per lo stesso numero esistono infinite rappresentazioni in notazione posizionale: ad esempio, $(10)_2$ indica il numero 2, allo stesso modo di $(010)_2$, $(0010)_2$, ecc., cioè 2 è un numero di n cifre in base 2, con $n \geq 2$. Nell'implementazione, per eliminare le ambiguità si può imporre che la cifra più significativa sia diversa da 0.

Lemma 3.1.2. La somma di due numeri $(u_{n-1} \dots u_1 u_0)_b$ e $(v_{n-1} \dots v_1 v_0)_b$, di n cifre ed in base b, è un numero di $n+1$ cifre esprimibile come

$$(w_n w_{n-1} \dots w_1 w_0)_b,$$

in cui chiamiamo w_n il carry ed il suo valore è 0 o 1.

Dimostrazione. Per definizione, $u < b^n$ e $v < b^n$ e, poiché $b \geq 2$,

$$u + v < 2b^n < b \cdot b^n = b^{n+1}.$$

Inoltre il carry può essere solo 0 o 1, perché

$$u + v < 2b^n \quad \iff \quad u + v \leq 1 \cdot b^n + (b^n - 1),$$

in cui l'ultimo addendo è il più grande numero di n cifre. □

[1] I limiti dell'implementazione sono discussi nella sezione 3.1.9.

Lemma 3.1.3. La moltiplicazione di due numeri di n e m cifre in base b $(u_{n-1} \dots u_1 u_0)_b$ e $(v_{,-1} \dots v_1 v_0)_b$ è un numero di $n+m$ cifre.

Dimostrazione. Sapendo che $u < b^n$ e $v < b^m$ la tesi è immediata, perché

$$u \cdot v < b^n \cdot b^m = b^{n+m}.$$

□

3.1.2 Introduzione agli algoritmi classici

Gli algoritmi fondamentali di aritmetica di base, anche chiamati algoritmi classici (si veda [3, p. 265]), sono:

a) addizione e sottrazione di due interi di n cifre, con risultato un intero di n cifre e un resto;

b) moltiplicazione di un intero di m cifre con uno di n cifre, con risultato un intero di $(m+n)$ cifre;

c) divisione di un intero di $(m+n)$ cifre per un intero di n cifre, con risultato un quoziente di $(m+1)$ cifre e un resto di n cifre.[2]

Per la scelta della base b è utile prendere la dimensione delle parole del calcolatore, per cui supponiamo di avere le seguenti operazioni di base:

a_0) addizione e sottrazione di due interi di 1 cifra, con risultato un intero di 1 cifra ed un resto;

b_0) moltiplicazione di due interi di 1 cifra, con risultato un intero di 2 cifre;

c_0) divisione di un intero di 2 cifre per uno di 1 cifra, a condizione che sia il quoziente sia il resto siano di 1 cifra.

Numeri di macchina

Per comodità si è scelta come base 2^{32}: numeri rappresentati da 32 o 64 bit dovrebbero essere supportati dalla maggior parte dell'hardware o dei compilatori. In C, per lavorare con interi di cui si deve essere sicuri della grandezza, si può utilizzare la libreria `inttypes.h` (che include a sua volta `stdint.h`, si vedano [5][6]), poiché nella definizione di standard del linguaggio la grandezza degli interi è lasciata all'implementazione. Perciò le cifre su cui lavoriamo sono `uint32_t`, interi da 0 a $2^{32}-1$, e `uint64_t`, da 0 a $2^{64}-1$.

[2] Per i nostri scopi non è necessaria la divisione di due interi di un qualunque numero di cifre ma è sufficiente che il divisore sia un numero di una sola cifra (la base scelta è sufficientemente grande perché l'operazione sia utile).

```
struct bignat {
        uint32_t  n; /* numero di cifre in base 2^32 */
        uint32_t *u; /* puntatore all'array di uint32_t */
};
```

Frammento 3.1: Implementazione della `struct bignat`.

Operazioni di base

L'aritmetica di macchina dei tipi interi e unsigned obbedisce alle leggi della aritmetica modulo 2^c, dove c è il numero di bit che compongono il tipo (si veda [4, p. 36]). Questo vuol dire che si verifica *wrap around* nel caso di overflow o underflow: per catturarli è sufficiente un confronto del risultato con uno degli operandi.

Quindi, dati `a` e `b` di tipo `uint32_t`, le operazioni di base sono:

a_0) `a+b` restituisce $(a+b) \mod 2^{32}$, mentre il carry è 1 se è vera la condizione `(a+b < a)` (o equivalentemente `(a+b < b)`), 0 altrimenti; per la sottrazione, `a-b` restituisce $(a-b) \mod 2^{32}$ e il borrow è -1 se `(a-b > a)`, 0 altrimenti;

b_0) per la moltiplicazione è utile ricorrere ad una soluzione di tipo `uint64_t` utilizzando cast espliciti, perciò il valore di `(uint64_t)a * (uint64_t)b` equivale a $a \cdot b$; se il risultato viene salvato nella variabile `prod`, si può ottenere la sua cifra di 32 bit meno significativa con `(uint32_t)prod`, quella più significativa con `(uint32_t)(prod>>32)`;

c_0) la divisione segue le regole inverse della moltiplicazione, sfruttando le operazioni di divisione intera `/` e di resto della divisione `%`.

3.1.3 Definizione

La struttura scelta per rappresentare naturali arbitrariamente grandi, nei limiti discussi nella sezione 3.1.9, è la seguente:

- un array di parole di 32 bit, ordinato dalla cifra meno significativa alla più significativa (ad esempio la prima parola dell'array, di indice 0, corrisponde alla cifra meno significativa, come mostrato in figura 3.1);
- un intero non negativo di 32 bit rappresentante la lunghezza dell'array.

L'implementazione è mostrata nel frammento 3.1.

Non tutte le possibili combinazioni di valori sono accettate: l'intero ovviamente deve indicare la grandezza dell'array e inoltre, per evitare spreco

```
0 static void normalize(struct bignat *u)
1 {
2         uint32_t j; /* cifra piu' significativa non nulla */
3
4         for (j = u->n - 1; u->u[j] == 0 && j > 0; j--) {}
5         u->n = j + 1;
6         u->u = realloc(u->u, sizeof(uint32_t)*(u->n));
7 }
```

Frammento 3.2: Implementazione di `normalize`.

di memoria e ambiguità nella rappresentazione, imponiamo che la cifra più significativa sia diversa da 0, compiendo un'eccezione per il numero zero.

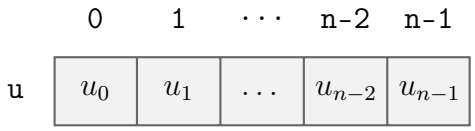

Figura 3.1: Struttura in memoria di $(u_{n-1}u_{n-2}\dots u_1u_0)_{2^{32}}$.

Dato $u \in \mathbb{N}$, il numero di cifre in base 2^{32} sufficienti per rappresentarlo è $\lfloor \log_{2^{32}} n \rfloor + 1$, cioè $\Theta(\log u)$.

`normalize`

Nell'implementazione delle operazioni aritmetiche tra `bignat`, si è scelto di allocare a priori la memoria massima necessaria per rappresentare i risultati, che non necessariamente rispetteranno le regole imposte per evitare ambiguità. La procedura `normalize`, mostrata nel frammento 3.2, elimina le cifre nulle più significative in eccesso di un `bignat` (caso speciale lo zero). Se usato sul risultato appena allocato di un'operazione, la sua complessità non influisce sulla complessità totale.

3.1.4 Addizione

In questa sezione si studia un algoritmo per la somma $(w_nw_{n-1}\dots w_1w_0)_b$ di due interi di n cifre $(u_{n-1}\dots u_1u_0)_b$ e $(v_{n-1}\dots v_1v_0)_b$. Nel caso in cui u e v non abbiano lo stesso numero di cifre, il minore viene trattato come un intero di tante cifre quante il maggiore.

```
0  struct bignat *bn_add(struct bignat *u, struct bignat *v)
1  {
2          char k; /* carry (0 o 1) */
3          struct bignat *w; /* risultato */
4
5          w = malloc(sizeof(struct bignat));
6          w->n = max(u->n,v->n) + 1;
7          w->u = malloc(sizeof(uint32_t)*w->n);
8
9          k = 0;
10         for (uint32_t j = 0; j < w->n - 1; j++) {
11                 w->u[j] = digit(u,j) + digit(v,j) + k;
12                 k = (w->u[j] < digit(u,j));
13         }
14         w->u[w->n - 1] = k;
15
16         normalize(w);
17         return w;
18 }
```

Frammento 3.3: Implementazione di `bn_add`.

Idea

L'algoritmo è una schematizzazione dell'addizione che si insegna alle elementari e scandisce gli addendi dalle cifre meno significative a quelle più significative, scrivendo una ad una le cifre del risultato e ricordandosi il carry (che per il lemma 3.1.2 è 0 o 1).

Implementazione

Nell'implementazione, mostrata dal frammento 3.3, `max` restituisce il maggiore di due `uint32_t`, mentre `digit` è una funzione che, dato w e un naturale i, restituisce il valore di w_i nella sua notazione in base 2^{32}, anche quando i supera la cifra più significativa non nulla.

Correttezza

Teorema 3.1.4. `bn_add`, dati gli interi non negativi di n cifre $(u_{n-1} \dots u_1 u_0)_b$ e $(v_{n-1} \dots v_1 v_0)_b$, restituisce la loro somma $(w_n w_{n-1} \dots w_1 w_0)_b$.

Dimostrazione. Il nucleo della dimostrazione è la dimostrazione di correttezza parziale del ciclo for.

precondizioni:

$$u, v \geq 0 \tag{3.1}$$
$$j = 0 \tag{3.2}$$
$$k = 0 \tag{3.3}$$

condizione per l'iterazione:

$$j < n \tag{3.4}$$

invarianti:

$$j \geq 0 \tag{3.5}$$
$$(kw_{j-1} \dots w_1 w_0)_b = (u_{j-1} \dots u_1 u_0)_b + (v_{j-1} \dots v_1 v_0)_b \tag{3.6}$$

postcondizione:

$$(kw_{n-1} \dots w_1 w_0)_b = (u_{n-1} \dots u_1 u_0)_b + (v_{n-1} \dots v_1 v_0)_b \tag{3.7}$$

1. **validità iniziale delle invarianti:** la (3.2) implica la (3.5) e la (3.6) è valida perché
$$(k)_b = (0)_b = 0 + 0,$$
per la (3.2) e la (3.3);

2. **conservazione nella generica iterazione:** una iterazione consiste nelle operazioni
   - $w_j = (u_j + v_j + k) \mod b$,
   - $k' = \lfloor (u_j + v_j + k)/b \rfloor$,
   - $j' = j + 1$;

   assumendo valide le precondizioni bisogna verificare che
$$(k'w_{j'-1} \dots w_1 w_0)_b = (u_{j'-1} \dots u_1 u_0)_b + (v_{j'-1} \dots v_1 v_0)_b, \tag{3.8}$$
mentre è banale che $j' = j+1 \geq 0$; u_j e v_j per la definizione di notazione posizionale sono compresi tra 0 e $b-1$ (inclusi), mentre k è pari a 0 o 1 perché carry di due numeri di j cifre per la (3.6), quindi
$$0 \leq u_j + v_j + k < 2b,$$
che si può distinguere nei seguenti due casi:

(a) se $0 \leq u_j + v_j + k < b$, allora
- $w_j = u_j + v_j + k$;
- $k' = 0$;
- $j' = j + 1$;

quindi

$$\begin{aligned}(3.8) &\iff (0w_{j'-1} \dots w_1 w_0)_b = (u_{j'-1} \dots u_1 u_0)_b + (v_{j'-1} \dots v_1 v_0)_b \\ &\iff (w_j \dots w_1 w_0)_b = (u_j \dots u_1 u_0)_b + (v_j \dots v_1 v_0)_b \\ &\iff (u_j + v_j) \cdot b^j + (kw_{j-1} \dots w_1 w_0)_b = u_j \cdot b^j + \\ &\quad + (u_{j-1} \dots u_1 u_0)_b + v_j \cdot b^j + (v_{j-1} \dots v_1 v_0)_b \\ &\iff (u_j + v_j) \cdot b^j = (u_j + v_j) \cdot b^j;\end{aligned}$$

(b) se $b \leq u_j + v_j + k < 2b$, allora la somma dei tre addendi si può scrivere come $b + r$, con $0 \leq r < b$ e
- $w_j = (b + r) \mod b = r$;
- $k' = \lfloor (b + r)/b \rfloor = 1$;
- $j' = j + 1$;

cioè

$$\begin{aligned}(3.8) &\iff (1w_{j'-1} \dots w_1 w_0)_b = (u_{j'-1} \dots u_1 u_0)_b + (v_{j'-1} \dots v_1 v_0)_b \\ &\iff (1w_j \dots w_1 w_0)_b = (u_j \dots u_1 u_0)_b + (v_j \dots v_1 v_0)_b \\ &\iff 1 \cdot b^{j+1} + (r - k) \cdot b^j + (kw_{j-1} \dots w_1 w_0)_b = \\ &\quad = (u_j + v_j) \cdot b^j + (u_{j-1} \dots u_1 u_0)_b + (v_{j-1} \dots v_1 v_0)_b \\ &\iff (b + r - k)b^j = (u_j + v_j)b^j \\ &\iff u_j + v_j + k - k = u_j + v_j;\end{aligned}$$

quindi le invarianti si conservano;

3. **validità della postcondizione:** la condizione per l'iterazione fallisce quando $j = n$, verificando la postcondizione (3.7).

Il ciclo termina sicuramente e la procedura assegna il valore k a w_n, rendendo $w = u + v$. □

Complessità

Per quanto riguarda la parte iniziale, è ragionevole supporre che la complessità di `malloc` sia lineare rispetto allo spazio da allocare. Il numero di iterazioni del ciclo for è $n + 1$, e ad ogni iterazione vengono eseguite un numero costante di operazioni aritmetiche di base: la complessità sia in tempo che in spazio della somma è $\Theta(n)$.

```
0  struct bignat *bn_sub(struct bignat *u, struct bignat *v)
1  {
2          /* u >= v */
3          signed char k; /* borrow (0 o -1) */
4          struct bignat *w; /* risultato */
5
6          w = malloc(sizeof(struct bignat));
7          w->n = max(u->n,v->n);
8          w->u = malloc(sizeof(uint32_t)*w->n);
9
10         k = 0;
11         for (uint32_t j = 0; j < u->n; j++) {
12                 w->u[j] = digit(u,j) - digit(v,j) + k;
13                 k = -(w->u[j] > digit(u,j));
14         }
15         /* k == 0 (perche' u >= v) */
16
17         normalize(w);
18         return w;
19 }
```

Frammento 3.4: Implementazione di `bn_sub`.

3.1.5 Sottrazione

In questa sezione si studia un algoritmo per la differenza $(w_{n-1} \dots w_1 w_0)_b$ di due interi di n cifre $(u_{n-1} \dots u_1 u_0)_b$ e $(v_{n-1} \dots v_1 v_0)_b$. Nel caso in cui u e v non abbiano lo stesso numero di cifre, il minore viene trattato come un intero di tante cifre quante il maggiore.

Idea

La differenza è molto simile all'addizione, con l'eccezione che le cifre si sottraggono e si tiene conto del prestito o borrow, anziché del carry. Perciò una richiesta aggiuntiva per gli argomenti è che il risultato della sottrazione debba essere positivo (verificabile dalla procedura `bn_cmp`), per assicurarsi che il prestito sia sempre possibile.

Implementazione

Nell'implementazione, mostrata dal frammento 3.4, `max` restituisce il maggiore di due `uint32_t`, mentre `digit` restituisce l'$(i+1)$-esima cifra meno significativa del `bignat` passato come argomento.

Complessità

Come la somma, la complessità è $\Theta(n)$ sia in tempo che in spazio.

3.1.6 Moltiplicazione

In questa sezione si studia un algoritmo per calcolare la moltiplicazione $(w_{n+m-1} \dots w_1 w_0)_b$ dei due interi $(u_{n-1} \dots u_1 u_0)_b$ e $(v_{m-1} \dots v_1 v_0)_b$ di rispettivamente n e m cifre.

Idea

Il metodo insegnato alle scuole primarie consiste nello scomporre la moltiplicazione di due interi di più cifre in una somma di moltiplicazioni del primo fattore per ogni cifra del secondo. Il seguente algoritmo segue gli stessi passaggi logici ma ottimizza lo spazio occupato: i risultati parziali sono accumulati in quello che sarà il risultato dell'operazione di moltiplicazione.

Implementazione

L'implementazione è mostrata dal frammento 3.5; il risultato allocato è di $n + m$ cifre, poiché esso è il numero massimo di cifre possibili per il lemma 3.1.3.

Correttezza

Lemma 3.1.5. Dati gli interi non negativi in base b di rispettivamente n, m e $n+m$ cifre $u = (u_{n-1} \dots u_1 u_0)_b$, $v = (v_{m-1} \dots v_1 v_0)_b$ e $w = (w_{n+m-1} \dots w_1 w_0)_b$ e dato l'intero $j \in \{0, 1, \dots, m-1\}$, l'esecuzione delle righe 14-23 è tale che

$$w' = u \cdot v_j b^j + w.$$

Dimostrazione. Il nucleo della dimostrazione è la dimostrazione di correttezza del ciclo for.

precondizioni:

$$u, v, w \geq 0 \tag{3.1}$$
$$k = 0 \tag{3.2}$$
$$0 \leq j \leq m - 1 \tag{3.3}$$
$$i = 0 \tag{3.4}$$

```
0  struct bignat *bn_mul(struct bignat *u, struct bignat *v)
1  {
2          struct bignat *w; /* risultato */
3          uint64_t t; /* risultato moltiplicazione di due cifre */
4          uint32_t k; /* cifra piu' significativa di t */
5
6          w = malloc(sizeof(struct bignat));
7          w->n = u->n + v->n;
8          w->u = malloc(sizeof(uint32_t)*w->n);
9
10         for (uint32_t i = 0; i < w->n; i++)
11                 w->u[i] = 0;
12
13         for (uint32_t j = 0; j < v->n; j++) {
14                 k = 0;
15                 for (uint32_t i = 0; i < u->n; i++) {
16                         t = (uint64_t) u->u[i] * (uint64_t) v->u[j] +
17                                 (uint64_t) w->u[i+j] + (uint64_t) k;
18                         w->u[i+j] = (uint32_t) t;
19                         k = (uint32_t) (t>>32);
20                 }
21                 w->u[j + u->n] = k;
22         }
23
24         normalize(w);
25         return w;
26 }
```

Frammento 3.5: Implementazione di `bn_mul`.

condizione per l'iterazione:

$$i < n \tag{3.5}$$

invarianti: sia $\hat{w} = (\hat{w}_{n+m-1} \dots \hat{w}_1 \hat{w}_0)_b$ pari a w all'inizio dell'esecuzione,

$$(kw_{i+j-1} \dots w_1 w_0)_b = (u_{i-1} \dots u_1 u_0)_b \cdot v_j b^j + (\hat{w}_{i+j-1} \dots \hat{w}_1 \hat{w}_0)_b \tag{3.6}$$

$$(w_{n+m-1} \dots w_{i+j+1} w_{i+j})_b = (\hat{w}_{n+m-1} \dots \hat{w}_{i+j+1} \hat{w}_{i+j})_b \tag{3.7}$$

postcondizione:

$$(kw_{n+j-2} \dots w_1 w_0)_b = u \cdot v_j b^j + \hat{w} \tag{3.8}$$

1. **validità iniziale delle invarianti:** per la (3.2) e la (3.4), e poiché siamo all'inizio dell'esecuzione,

$$(0 w_j \dots w_1 w_0)_b = (0)_b \cdot v_j b^j + (\hat{w}_j \dots \hat{w}_1 \hat{w}_0)_b,$$
$$(w_{n+m-1} \dots w_{j+2} w_j)_b = (\hat{w}_{n+m-1} \dots \hat{w}_{j+2} \hat{w}_j)_b;$$

2. **conservazione nella generica iterazione:** una iterazione consiste nelle seguenti operazioni:

   - $w'_{i+j} = (u_i \cdot v_j + w_{i+j} + k) \mod b$;
   - $k' = \left\lfloor (u_i \cdot v_j + w_{i+j} + k)/b \right\rfloor$;
   - $i' = i + 1$;

   assumendo valide le invarianti, bisogna verificare che

$$(k' w'_{i'+j-1} \dots w_1 w_0)_b = (u_{i'-1} \dots u_1 u_0)_b \cdot v_j b^j + (\hat{w}_{i'+j-1} \dots \hat{w}_1 \hat{w}_0)_b \tag{3.9}$$

$$(w_{n+m-1} \dots w_{i'+j+2} w_{i'+j})_b = (\hat{w}_{n+m-1} \dots \hat{w}_{i'+j+2} \hat{w}_{i'+j})_b \tag{3.10}$$

   la (3.10) è banalmente verificata, perché l'unica cifra modificata di w è quella di indice $i + j$, invece è facile notare come

$$k' \cdot b + w'_{i+j} = u_i \cdot v_j + w_{i+j} + k,$$

   utile nella seguente catena di doppie implicazioni

$$\begin{aligned}
(9) &\iff (k' w'_{i+j} \dots w_1 w_0)_b = (u_i \dots u_1 u_0)_b \cdot v_j b^j + (\hat{w}_{i+j} \dots \hat{w}_1 \hat{w}_0)_b \\
&\iff k' b^{i+j+1} + w'_{i+j} b^{i+j} + (w_{i+j-1} \dots w_1 w_0)_b = \\
&\quad u_i v_j b^{i+j} + (u_{i-1} \dots u_1 u_0)_b \cdot v_j b^j + \hat{w}_{i+j} b^{i+j} + (\hat{w}_{i+j-1} \dots \hat{w}_1 \hat{w}_0)_b \\
&\iff k' b^{i+j+1} + w'_{i+j} b^{i+j} - k b^{i+j} + (k w_{i+j-1} \dots w_1 w_0)_b = \\
&\quad u_i v_j b^{i+j} + (u_{i-1} \dots u_1 u_0)_b \cdot v_j b^j + \hat{w}_{i+j} b^{i+j} + (\hat{w}_{i+j-1} \dots \hat{w}_1 \hat{w}_0)_b \\
&\iff k' b^{i+j+1} + w_{i+j} b^{i+j} = u_i v_j b^{i+j} + \hat{w}_{i+j} b^{i+j} + k b^{i+j} \\
&\iff kb + w'_{i+j} = u_i v_j + \hat{w}_{i+j} + k,
\end{aligned}$$

che è vera perché $\hat{w}_{i+j} = w_{i+j}$ per l'invariante (3.7).

3. **validità della postcondizione:** la condizione per l'iterazione fallisce quanto $i = n$, che verifica la postcondizione (3.8).

Il ciclo termina sicuramente e dopo la sua esecuzione viene assegnato a w_{n+j-1} il valore di k, rendendo valida la tesi. □

Teorema 3.1.6. Dati $u = (u_{n-1} \dots u_1 u_0)_b$ e $v = (v_{m-1} \dots v_1 v_0)_b$, `bn_mul` restituisce $w = u \cdot v$.

Dimostrazione. Per la proprietà distributiva della moltiplicazione rispetto alla somma

$$u \cdot v = \sum_{j=0}^{m-1} u \cdot v_j b^j = u \cdot v_0 b^0 + u \cdot v_1 b^1 + \cdots + u \cdot v_{m-1} b^{m-1},$$

che è esattamente il valore calcolato dal ciclo for esterno di `bn_mul` dopo aver inizializzato w a 0, per quanto dimostrato dal lemma 3.1.5. □

Complessità

Il ciclo for esterno scandisce le cifre di v, il ciclo for interno quelle di u: la complessità in tempo è $\Theta(n \cdot m)$, quella in spazio $\Theta(n + m)$.

3.1.7 Divisone per una cifra

In questa sezione si studia un algoritmo per il calcolo della divisione intera $(w_{n-1} \dots w_1 w_0)_b$ di un intero non negativo $(u_{n-1} \dots u_1 u_0)_b$ per un intero di una cifra $(a)_b$.

Idea

Poiché il divisore è di una cifra, è sufficiente scandire il dividendo dalla cifra più significativa alla meno significativa, trovando la cifra corrispondente del risultato della divisione e ricordando ad ogni passo il resto della divisione.

Implementazione

L'implementazione è mostrata nel frammento 3.6.

```
0  struct bignat *bn_div_uint32(struct bignat *u, uint32_t a)
1  {
2          struct bignat *v; /* risultato */
3          uint64_t t; /* dividendo parziale */
4          uint32_t r; /* resto parziale */
5
6          v = malloc(sizeof(struct bignat));
7          v->n = u->n;
8          v->u = malloc(sizeof(uint32_t)*v->n);
9
10         r = 0;
11         for (uint32_t i = u->n - 1; i < UINT32_MAX; i--) {
12                 t = ((uint64_t)r<<32) + u->u[i];
13                 v->u[i] = (uint32_t) (t / a);
14                 r = (uint32_t) (t % a);
15         }
16
17         normalize(v);
18         return v;
19 }
```

Frammento 3.6: Implementazione di `bn_div_uint32`.

Correttezza

`bn_div_uint32`, dati $u = (u_{n-1} \dots u_1 u_0)_b$ e $a \in \mathbb{N}$ tale che $a < b$, restituisce $v = (v_{n-1} \dots v_1 v_0)_b$ per cui vale $v = \lfloor u/a \rfloor$.

Dimostrazione. Il nucleo della dimostrazione è la dimostrazione di correttezza del ciclo for.

precondizioni:

$$u \geq 0 \tag{3.1}$$
$$0 \leq a \leq b - 1 \tag{3.2}$$
$$i = n - 1 \tag{3.3}$$
$$r = 0 \tag{3.4}$$

condizione per l'iterazione:

$$i \geq 0 \tag{3.5}$$

invarianti:

$$0 \leq r \leq b - 1 \tag{3.6}$$
$$(u_{n-1} \dots u_{i+2} u_{i+1})_b = (v_{n-1} \dots v_{i+2} v_{i+1})_b \cdot a + r \tag{3.7}$$

postcondizioni:

$$0 \leq r \leq b - 1 \tag{3.8}$$
$$(u_{n-1} \dots u_1 u_0)_b = (v_{n-1} \dots v_1 v_0)_b \cdot a + r \tag{3.9}$$

1. **validità iniziale delle invarianti:** la (3.4) implica la (3.6) e la (3.7) è valida, perché per la (3.3) e la (3.4) $0 = 0 \cdot a + 0$;

2. **conservazione nella generica iterazione:** la generica iterazione effettua le seguenti operazioni:

   - $v_i = \lfloor (r \cdot b + u_i)/a \rfloor$;
   - $r' = (r \cdot b + u_i) \mod a$;
   - $i' = i - 1$;

   assumendo valide le invarianti, va verificato che

   $$0 \leq r' \leq b - 1 \tag{3.10}$$
   $$(u_{n-1} \dots u_{i'+2} u_{i'+1})_b = (u_{n-1} \dots u_{i'+2} u_{i'+1})_b \cdot a + r' \tag{3.11}$$

   la (3.10) è facilmente verificata, poiché per la (3.2) $r' < a < b$, mentre notare che

   $$r \cdot b + u_i = v_i \cdot a + r'$$

   è utile nella seguente catena di doppie implicazioni

   $$\begin{aligned}
   (3.11) &\iff (u_{n-1} \dots u_{i+1} u_i)_b = (v_{n-1} \dots v_{i+1} v_i)_b \cdot a + r' \\
   &\iff (u_{n-1} \dots u_{i+2} u_{i+1})_b \cdot b + u_i = \\
   &\quad (v_{n-1} \dots v_{i+2} v_{i+1})_b \cdot b \cdot a + v_i \cdot a + r' \\
   &\iff (u_{n-1} \dots u_{i+2} u_{i+1})_b \cdot b = (v_{n-1} \dots v_{i+2} v_{i+1})_b \cdot b \cdot a + r \cdot b \\
   &\iff (u_{n-1} \dots u_{i+2} u_{i+1})_b = (v_{n-1} \dots v_{i+2} v_{i+1})_b \cdot a + r,
   \end{aligned}$$

   e quest'ultima uguaglianza è esattamente l'invariante (3.7).

3. **validità della postcondizione:** la condizione per l'iterazione fallisce quando $i = -1$, così verificando la postcondizione (3.9); la postcondizione (3.8) è garantita dall'invariante (3.6).

Le due postcondizioni del ciclo for verificano che v è la divisione intera di u per a, con resto r. □

Complessità

`bn_div_uint32` scandisce le cifre di u, eseguendo un numero finito di istruzioni ad ogni iterazione del ciclo for: la complessità sia in tempo che in spazio è $\Theta(n)$.

3.1.8 Altre procedure utili

Oltre alle procedure per la creazione, la stampa e la distruzione di `bignat`, altre procedure utili implementate sono:

- `bn_bits`, che restituisce il minor numero di cifre necessarie per rappresentare il `bignat` passato come argomento in base 2;
- `bn_get`, che dato un `bignat` e un indice i, restituisce l'$(i+1)$-esima cifra più significativo della rappresentazione del `bignat` in base 2;
- `bn_cmp`, che dati due `bignat` restitusisce 0 se rappresentano lo stesso intero non negativo, 1 se il primo argomento è maggiore del secondo e -1 altrimenti;
- `bn_iszero`, che restituisce 1 se il `bignat` passatogli come argomento rappresenta lo 0, 0 altrimenti;
- `bn_rem_uint32` e `bn_remcheck_uint32`, che dati un `bignat` e una cifra i, restituiscono rispettivamente il resto e 1 o 0, a seconda che il resto sia diverso da 0, della divisione intera del `bignat` per i;
- `bn_rshift` e `bn_lshift` (effettivamente implementate nella libreria per i `bigfloat`), dati un `bignat` e un intero non negativo i, eseguono rispettivamente la divisione e la moltiplicazione del `bignat` per $(2^{32})^i$.

3.1.9 Limitazioni

Per come i `bignat` sono stati costruiti, essi non sono letteralmente grandi a piacere, nonostante la loro grandezza sia variabile: oltre alla disponibilità della memoria di un calcolatore, la grandezza dell'array è limitata perché indicata da un numero di macchina di 32 bit, cioè il massimo `bignat` rappresentabile è un array di `uint32_t` lungo $2^{32} - 1$, cioè di $2^{37} - 32$ bit (circa 16 GiB) che corrisponde all'intero $2^{2^{37}-32} - 1$.

```c
struct bigfloat {
        signed char sign; /* 1 o -1 */
        struct bignat *m; /* mantissa */
        int64_t e; /* esponente */
};
```

Frammento 3.7: Implementazione della `struct bigfloat`.

3.2 Razionali precisi a piacere: `bigfloat`

In questa sezione vengono costruiti dei numeri razionali a precisione arbitraria[3] a partire dai naturali a grandezza arbitraria, salvati in memoria in base 2^{32}.

3.2.1 Definizione

Per rendere semplice la scrittura e il calcolo delle complessità, è stata scelta una rappresentazione di numeri razionali sotto forma di segno, esponente e mantissa: il generico razionale cioè è

$$s \cdot m \cdot (b)^e \quad \text{con} \quad s \in \{-1, 1\},\ m, b \in \mathbb{N},\ e \in \mathbb{Z},$$

in cui s è il segno, m la mantissa, b la base ed e l'esponente.

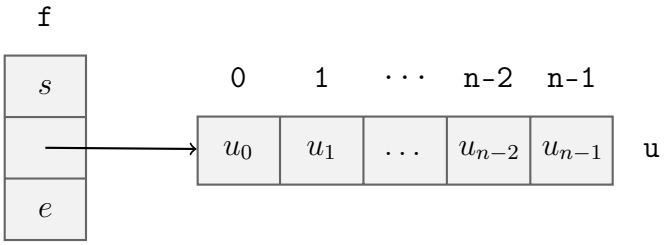

Figura 3.2: Struttura in memoria di $f = s \cdot (u_{n-1}u_{n-2}\dots u_1u_0)_{2^{32}} \cdot (2^{32})^e$.

Nell'implementazione in C si è scelto di utilizzare come segno un `char`, come esponente un `int64_t` e come mantissa un `bignat`: una scelta obbligata per la base è la stessa dei `bignat`, 2^{32}, per evitare ambiguità nella rappresentazione. Inoltre si pone la seguente condizione per una rappresentazione valida di un razionale: la cifra meno significativa (in base 2^{32}) di m deve essere diversa da 0 e per convenzione lo zero ha segno positivo ed esponente pari a 0.

[3] I limiti dell'implementazione sono discussi nella sezione 3.8.

```
0  static void normalize(struct bigfloat *f)
1  {
2          unsigned long int i; /* indice cifra meno significativa non nulla */
3          struct bignat *tbn;
4
5          if (bn_iszero(f->m)) { /* controllo zero */
6                  bn_destroy(f->m);
7                  f->sign = 1;
8                  f->m = bn_zero();
9                  f->e = 0;
10                 return;
11         }
12
13         for (i = 0; f->m->u[i] == 0 && i < f->m->n; i++) {}
14
15         if (i > 0) {
16                 tbn = f->m;
17                 f->m = bn_rshift(f->m, i);
18                 bn_destroy(tbn);
19
20                 f->e += i;
21         }
22 }
```

Frammento 3.8: Implementazione di `normalize`.

`normalize`

Il risultato di un'operazione aritmetica potrebbe non rispettare la regola appena stabilita: la procedura `normalize` modifica il `bigfloat` passatogli come argomento mantenendo invariato il valore, cioè tale che

$$s' \cdot m' \cdot (2^{32})^{e'} = s \cdot m \cdot (2^{32})^{e},$$

ma in modo che la cifra meno significativa della mantissa sia diversa da 0. La complessità di `normalize` non influisce nelle operazioni in cui un risultato appena creato viene normalizzato, poiché è asintoticamente minore o uguale della complessità della sua creazione.

3.2.2 Addizione

In questa sezione si studia un algoritmo per la addizione h di due `bigfloat` $f = s_f m_f (2^{32})^{e_f}$ e $g = s_g m_g (2^{32})^{e_g}$.

Idea

Per ottenere la somma vanno trovate le mantisse di $f' = f$ e $g' = g$ con $e_{f'} = e_{g'} = \min(e_f, e_g)$. Poi è sufficiente calcolare la somma $s'_f m'_f + s'_g m'_g$ per poter scrivere il risultato.

Implementazione

L'implementazione è mostrata nel frammento 3.9.

Accenno di correttezza

L'algoritmo calcola le mantisse u e v, corrispondenti a m_f e m_g moltiplicate per una potenza di 2^{32}, tali che

$$\begin{aligned} f + g &= s_f \cdot m_f \cdot (2^{32})^{e_f} + s_g \cdot m_g \cdot (2^{32})^{e_g} \\ &= s_f \cdot u \cdot (2^{32})^{\min(e_f, g_f)} + s_g \cdot v \cdot (2^{32})^{\min(e_f, g_f)} \\ &= (s_f \cdot u + s_g \cdot v) \cdot (2^{32})^{\min(e_f, g_f)}. \end{aligned}$$

Complessità

L'algoritmo deve trovare e sommare due `bignat` la cui cifra meno significativa (in base 2^{32}) rappresenta la potenza di 2^{32} con esponente $\min(e_f, e_g)$ e la cui cifra più significativa rappresenta la potenza con esponente $\max(e_f + \lfloor \log_{2^{32}} m_f \rfloor, e_g + \lfloor \log_{32} m_g \rfloor)$. La complessità sia in tempo che in spazio è lineare alla differenza

$$\max(e_f + \lfloor \log_{2^{32}} m_f \rfloor, e_g + \lfloor \log_{32} m_g \rfloor) - \min(e_f, e_g).$$

3.2.3 Sottrazione

In questa sezione si studia un algoritmo per la sottrazione h di due `bigfloat` $f = s_f m_f (2^{32})^{e_f}$ e $g = s_g m_g (2^{32})^{e_g}$.

Idea

Poiché l'addizione tratta già argomenti di segno sia positivo che negativo, la procedura `bf_sub` si appoggia appropriatamente a `bf_add`.

Implementazione

L'implementazione è mostrata nel frammento 3.10.

```
0  struct bigfloat *bf_add(struct bigfloat *f, struct bigfloat *g)
1  {
2          struct bigfloat *h; /* risultato */
3          struct bignat *u, *v; /* addendi parziali */
4
5          h = malloc(sizeof(struct bigfloat));
6
7          if (f->e < g->e) {
8                  u = bn_copy(f->m);
9                  v = bn_lshift(g->m, g->e - f->e);
10                 h->e = f->e;
11         } else if (f->e > g->e) {
12                 u = bn_lshift(f->m, f->e - g->e);
13                 v = bn_copy(g->m);
14                 h->e = g->e;
15         } else {
16                 u = bn_copy(f->m);
17                 v = bn_copy(g->m);
18                 h->e = f->e;
19         }
20
21         if (f->sign == g->sign) {
22                 h->m = bn_add(u,v);
23                 h->sign = f->sign;
24         } else if (f->sign == 1) { /* g->sign == -1 */
25                 if (bn_cmp(u,v) >= 0) {
26                         h->m = bn_sub(u,v);
27                         h->sign = +1;
28                 } else {
29                         h->m = bn_sub(v,u);
30                         h->sign = -1;
31                 }
32         } else { /* f->sign == -1 && g->sign == 1 */
33                 if (bn_cmp(u,v) < 0) {
34                         h->m = bn_sub(v,u);
35                         h->sign = +1;
36                 } else {
37                         h->m = bn_sub(u,v);
38                         h->sign = -1;
39                 }
40         }
41
42         normalize(h);
43         bn_destroy(u);
44         bn_destroy(v);
45         return h;
46 }
```

Frammento 3.9: Implementazione di `bf_add`.

```
0  struct bigfloat *bf_sub(struct bigfloat *f, struct bigfloat *g)
1  {
2          struct bigfloat *minusg;
3          struct bigfloat *res;
4
5          if (bn_iszero(g->m))
6                  return bf_copy(f);
7
8          minusg = bf_copy(g);
9          minusg->sign *= -1; /* cambio segno */
10
11         res = bf_add(f, minusg);
12
13         bf_destroy(minusg);
14         return res;
15 }
```

Frammento 3.10: Implementazione di `bf_sub`.

Accenno di correttezza

La procedura `bn_sub` chiama `bn_add` con argomenti f e $-g$, poiché banalmente

$$f - g = f + (-g).$$

Complessità

La complessità equivale a quella dell'addizione sia in tempo che in spazio.

3.2.4 Moltiplicazione

In questa sezione si studia un algoritmo per la moltiplicazione h di due `bigfloat` $f = s_f m_f (2^{32})^{e_f}$ e $g = s_g m_g (2^{32})^{e_g}$.

Idea

Sfruttando l'operazione `bn_mul` offerta dai `bignat`, la moltiplicazione è piuttosto semplice.

Implementazione

L'implementazione è mostrata nel frammento 3.11.

```
0  struct bigfloat *bf_mul(struct bigfloat *f, struct bigfloat *g)
1  {
2          struct bigfloat *h; /* risultato */
3
4          if (bn_iszero(f->m) || bn_iszero(g->m))
5                  return bf_zero();
6
7          h = malloc(sizeof(struct bigfloat));
8          h->sign = f->sign * g->sign;
9          h->m = bn_mul(f->m,g->m);
10         h->e = f->e + g->e;
11
12         normalize(h);
13         return h;
14 }
```

Frammento 3.11: Implementazione di `bf_mul`.

Accenno di correttezza

Per la proprietà commutativa della moltiplicazione

$$f \cdot g = s_f \cdot m_f \cdot (2^{32})^{e_f} \ \cdot \ s_g \cdot m_g \cdot (2^{32})^{e_g} = (s_f \cdot s_g) \cdot (m_f \cdot m_g) \cdot (2^{32})^{e_f + e_g},$$

che è esattamente il risultato calcolato da `bf_mul`.

Complessità

La procedura esegue un numero di operazioni finite, ad eccezione della moltiplicazione tra `bignat`: se n_f e n_g sono le cifre delle mantisse di f e g, la complessità in tempo della moltiplicazione, che indichiamo come $M(n_f, n_g)$ è tale che

$$M(n_f, n_g) = \Theta(n_f \cdot n_g),$$

mentre la complessità in spazio è $O(n_f + n_g)$.

3.2.5 Approssimazione

In questa sezione si studiano le operazioni di approssimazione per difetto e per eccesso di un `bigfloat` $f = s_f \cdot m_f \cdot (2^{32})^{e_f}$ ad una data potenza della base con esponente $d \in \mathbb{Z}$. È importante prendere in considerazione il segno: se f è negativo, l'approssimazione per difetto deve essere minore o uguale a f e l'approssimazione per eccesso deve essere maggiore o uguale.

Idea

Dato un razionale, nel caso questo sia positivo, l'approssimazione h alla potenza $(2^{32})^d$ per difetto consiste nel troncare la mantissa e cambiare l'esponente, se necessario, in modo che e_h sia maggiore o uguale a d; l'approssimazione per eccesso alla potenza $(2^{32})^d$ invece tronca e aggiunge uno alla mantissa troncata (se necessario) in modo che e_h sia maggiore o uguale a d. Se il razionale è negativo, il ragionamento è invertito.

Implementazione

L'implementazione di `bf_trunc` e `bf_round` è mostrata nei frammenti 3.12 e 3.13. È importante notare come il codice di `bf_trunc` si occupi di troncare la mantissa, quello di `bf_round` di troncarla e aggiungere uno (se necessario per approssimarla per eccesso): nel caso l'argomento sia negativo, si chiamano l'una con l'altra per ottenere l'approssimazione desiderata.

Accenno di correttezza

Per il caso positivo, se la cifra meno significativa di f supera la precisione richiesta, `bf_trunc` tronca e `bf_round` tronca e aggiunge uno: quest'ultima è corretta perché si suppone che la rappresentazione di f sia normalizzata, cioè che la sua cifra meno significativa sia non nulla.

Complessità

Le due procedure consistono di qualche semplice operazione lineare su `bignat`: la complessità, sia in tempo che in spazio, è $O(n_f)$, dove n_f è il numero di cifre della mantissa di f.

3.2.6 Divisone approssimata per un intero di una cifra

In questa sezione si studia il risultato g della divisione di un `bigfloat` $f = s_f \cdot m_f \cdot (2^{32})^{e_f}$ per i, approssimato fino alla potenza $(2^{32})^d$. Le seguenti funzioni sono necessarie perché il risultato di una divisione, anche per un intero, non è sempre rappresentabile in una data base con un numero finito di cifre.

Idea

Le due funzioni `bf_div_uint32_trunc` e `bf_div_uint32_round` aprossimano il risultato della divisione rispettivamente per difetto e per eccesso: per approssimare per eccesso si porta f ad avere come esponente $d-1$, una cifra in

```
0  struct bigfloat *bf_trunc(struct bigfloat *f, long long int d)
1  {
2          struct bignat *tbn;
3          struct bigfloat *res, *tbf;
4
5          res = bf_copy(f);
6
7          if (d <= f->e)
8                  return res;
9
10         if (res->sign == 1) {
11                 tbn = res->m;
12                 res->m = bn_rshift(res->m, llabs(d - res->e));
13                 bn_destroy(tbn);
14
15                 res->e = d;
16
17                 normalize(res);
18                 return res;
19         } else { /* per difetto */
20                 res->sign = 1;
21
22                 tbf = res;
23                 res = bf_round(res, d);
24                 bf_destroy(tbf);
25
26                 res->sign = -1;
27
28                 return res;
29         }
30 }
```

Frammento 3.12: Implementazione di `bf_trunc`.

```
0  struct bigfloat *bf_round(struct bigfloat *f, long long int d)
1  {
2          struct bignat *tbn;
3          struct bigfloat *res, *tbf;
4
5          res = bf_copy(f);
6
7          if (f->e >= d)
8                  return res;
9
10         if (res->sign == 1) {
11                 tbn = res->m;
12                 res->m = bn_rshift(res->m, llabs(res->e - d));
13                 bn_destroy(tbn);
14
15                 res->e = d;
16
17                 tbn = res->m;
18                 res->m = bn_succ(res->m);
19                 bn_destroy(tbn);
20
21                 normalize(res);
22                 return res;
23         } else { /* per eccesso */
24                 res->sign = 1;
25
26                 tbf = res;
27                 res = bf_trunc(res, d);
28                 bf_destroy(tbf);
29
30                 res->sign = -1;
31
32                 return res;
33         }
34 }
```

Frammento 3.13: Implementazione di `bf_round`.

più del necessario, per poi usare la divisione `bn_div_uint32` fornita dai `bignat` e le procedure `bf_trunc` e `bf_round`.

Correttezza

La correttezza di `bf_div_uint32_round` verge sul fatto che un eventuale resto nella divisione compare come una cifra meno significativa maggiore di 0 nel risultato di `bf_div_uint32` e viene catturata da `bf_trunc`.

3.2.7 Altre procedure utili

Oltre alle procedure per la creazione, la copia, la stampa e la distruzione di `bigfloat`, altre procedure utili implementate sono:

- `bf_cmp`, che dati due `bigfloat`, restituisce 0 se rappresentano lo stesso razionale, 1 se il primo argomento è maggiore del secondo, -1 altrimenti;
- `bf_tobignat` e `bf_frac`, che restituiscono rispettivamente la parte intera (come `bignat`) e la parte frazionaria di un numero razionale;
- `bf_rshift` e `bf_lshift`, che dato un `bigfloat` f e un intero non negativo d, restituiscono il `bigfloat` rappresentante rispettivamente la divisione e la moltiplicazione di f per $(2^{32})^d$;
- `bf_div_pow2_bn`, che dati il `bigfloat` f ed il `bignat` u, restituisce il risultato di $f/2^u$.

3.2.8 Limitazioni

Per come i `bigfloat` sono stati implementati, essi non sono letteralmente precisi a piacere, nonostante la loro grandezza sia variabile. Oltre alla disponibilità della memoria di un calcolatore, la precisione di un `bigfloat` è limitata dalla limitazione dei `bignat` e da quella degli interi di 64 cifre: il massimo numero in valore assoluto rappresentabile è

$$\left(2^{2^{37}-32}-1\right)\cdot(2^{32})^{2^{63}-1},$$

mentre il più piccolo è $1\cdot(2^{32})^{-(2^{63}-1)}$.

3.3 Lista doppiamente concatenata

È stata progettata una lista doppiamente concatenata che supporta le operazioni di creazione, inserimento in coda, scansione (dalla testa o dalla coda), cancellazione di un elemento qualsiasi e distruzione.

3.3.1 Descrizione

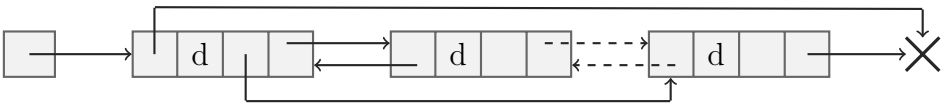

Figura 3.3: Rappresentazione di una lista.

Ogni elemento della lista doppiamente concatenata contiene:

- il puntatore al precedente `prev`;
- il puntatore ai dati satellite;
- il puntatore all'ultimo elemento della lista `last` (aggiornato solo in testa);
- il puntatore al successivo `next`;

La lista è rappresentata dal puntatore al suo elemento in testa, mentre `NULL` indica la lista vuota o un elemento vuoto.

Le liste in cui i dati sono vertici del grafo di appartenenza (`struct memnode`) sono di tipo `struct dl_list_n`, quelle in cui i dati sono interi non negativi (`struct bignat`) sono di tipo `struct dl_list_b`.

Operazioni

Seguono le operazioni principali sulle liste:

- `create` crea e restituisce una lista vuota (`NULL`);
- `add`, data una lista `lst` e il puntatore a dei dati satellite, crea un nuovo elemento della lista che punta ai dati e lo aggiunge in coda a `lst`; restituisce il puntatore alla nuova lista modificata;
- `remove`, data una lista `lst` ed un suo elemento `el`, rimuove l'elemento dalla lista e restituisce il puntatore alla lista aggiornata;

```
struct data *d; /* dati correnti */
struct list *scan; /* elemento corrente della lista */

scan = L;
while (!isempty(scan)) {
        d = get(scan);

        /* operazioni sui dati d */

        scan = next(scan);
}
```

Frammento 3.14: Esempio di scansione di una lista.

- `get` restituisce il puntatore ai dati satellite e `set` dato un elemento della lista ed il puntatore ad un dato del tipo corretto ne cambia i dati satellite;
- `isempty`, `ishead` e `istail` controllano se l'elemento dato rappresenta rispettivamente la lista vuota (`NULL`), la testa di una lista, la coda di una lista;
- le procedure `next`, `prev` e `last` dato un elemento restituiscono il puntatore all'elemento successivo, precedente, e in coda alla lista (`NULL` se non esiste).

Mentre una scansione di una lista `L` dalla testa alla coda può essere effettuata come mostrato nel frammento 3.14.

3.3.2 Complessità

Tutte le operazioni hanno complessità in tempo $\Theta(1)$, poiché contengono un numero finito di alternative if e operazioni di base di complessità costante, quindi l'esecuzione di p operazioni richiede tempo $\Theta(p)$. Inoltre gli elementi delle liste occupano uno spazio costante per elemento, quindi una lista di q elementi richiede $\Theta(q)$ spazio in memoria.

3.4 Grafo di appartenenza

Per rappresentare il grafo orientato di appartenenza di un insieme ereditariamente finito è stata scelta la struttura delle liste di adiacenza. Più in particolare, la `struct memgraph` contiene, come mostrato dal frammento 3.15:

- la lista dei vertici;

```
struct memgraph {
        struct dl_list_n *V; /* lista dei vertici */
        struct dl_list_n *xx; /* primi figli */
        struct interval *rcode; /* codice R_A */
};
```

Frammento 3.15: Implementazione della `struct memgraph`.

```
struct memnode {
        struct bignat *code; /* codice di Ackermann */

        int indegree; /* numero di genitori */
        int outdegree; /* numero di figli */
        struct dl_list_n *adj; /* lista di adiacenza */

        struct interval *rcode; /* approssimazione R_A */
        struct interval *rsingleton; /* approssimazione R_A singoletto */
};
```

Frammento 3.16: Implementazione della `struct memnode`.

- i vertici corrispondenti ai codici inizialmente caricati;
- l'eventuale approssimazione, di tipo `struct interval`, della codifica secondo $\mathbb{R}_A$ dell'insieme composto dai vertici iniziali.

Invece ogni vertice, di tipo `struct memnode`, la cui implementazione è mostrata nel frammento 3.16, contiene:

- il proprio codice di Ackermann (`struct bignat`);
- la propria lista di adiacenza (lista doppiamente concatenata di puntatori ad altri `struct node`, di tipo `struct dl_list_n`);
- il numero di figli;
- il numero di genitori;
- l'eventuale approssimazione, di tipo `struct interval`, della codifica secondo $\mathbb{R}_A$;
- l'eventuale approssimazione della codifica secondo $\mathbb{R}_A$ per il singoletto di questo nodo, giustificato dal teorema 2.4.1.

```
1 struct interval{
2         char incl; /* 1 se estremi inclusi, 0 se esclusi */
3         struct bigfloat *min;
4         struct bigfloat *max;
5 };
```

Frammento 3.17: Implementazione della `struct interval`.

3.5 Intervallo di razionali

Come viene osservato nel capitolo 5, è utile rappresentare dei dati affetti da errore come degli intervalli di numeri razionali. In tale capitolo vengono anche discusse le operazioni su intervalli.

3.5.1 Descrizione

La `struct interval` contiene, come mostrato nel frammento 3.17:

- gli estremi razionali ordinati (`struct bigfloat`);
- un parametro `incl` (`char`) che indica se l'intervallo è aperto o chiuso (equivalente al confronto degli estremi utilizzando `bf_cmp`).

Capitolo 4

Dalla codifica al grafo

Questo capitolo parla degli algoritmi per il calcolo del grafo di appartenenza di h_n, dato $n \in \mathbb{N}$, e la loro implementazione in C.

4.1 Input e inizializzazione

Per quanto stabilito dalle specifiche di input nella sezione 1.1, da *standard input* vengono passati i caratteri della rappresentazione binaria di n. Questa fase contiene anche parte della computazione, perché al posto di salvare l'input vengono salvati gli indici dei bit pari a 1, trovando così la lista dei codici dei figli di h_n, sufficienti per costruire il suo grafo di appartenenza e poi per approssimare $\mathbb{R}_A(h_n)$.

4.1.1 Idea

Le cifre binarie di n vengono lette dalla più significativa alla meno significativa, fino ad esaurimento, ma non ne si conosce la quantità a priori: l'algoritmo di lettura usa un `bignat` come indice per contare e salvare in una lista gli indici di arrivo dei bit pari a 1. Una volta terminata la lettura, a ciascuno di questi indici viene sottratto il numero di caratteri letti meno uno per ottenere gli indici corretti.

La lista dei codici dei figli di h_n viene poi usata per inizializzare il grafo di appartenenza.

4.1.2 Implementazione

La procedura `dllb_scanchildren`, la cui implementazione è mostrata nel frammento 4.1, legge n e restituisce la lista dei codici dei figli di h_n, ordinati

dal più grande al più piccolo. Invece `memgraph_create`, mostrata nel frammento 4.2, riceve questa lista come argomento e crea il grafo con i nodi iniziali dei figli di h_n.

4.1.3 Accenno di correttezza

Una conseguenza della definizione di $\mathbb{N}_A$ e della sua biettività è che $x \in h_n$ se e solo se il $(\mathbb{N}_A(x)+1)$-esimo bit di $\mathbb{N}_A(h_n)$ è 1. La procedura sfrutta questa proprietà e trova la lista decrescente dei codici dei figli.

4.1.4 Complessità

Sia d il numero dei caratteri rappresentanti le cifre binarie di n letti in input. La procedura `dllb_scanchildren`:

1. utilizza un `bignat` per contare da 0 a $d-1$, eventualmente salvando quelli che corrispondo ad un bit 1;

2. calcola il risultato della sottrazione di $d-1$ con tutti i `bignat` salvati.

Quindi, dati $c_0, \dots c_{d-1}$ le costanti per la gestione delle operazioni del contatore con $c_{\max}$ e $c_{\min}$ il loro massimo e minimo, la complessità di 1. in tempo e, nel caso peggiore in spazio, è

$$\begin{aligned} c_0 + c_1 + c_2 \log 2 + \cdots + c_{d-1} \log(d-1) &< c_{\max} \cdot \log\big((d)!\big) \\ &> c_{\min} \cdot \log\big((d-1)!\big), \end{aligned}$$

e poiché $(m/2)^{m/2} \leq (m)! \leq m^m$ per ogni $m \in \mathbb{N}$ la complessità è $\Theta(d \log d)$. Invece il punto 2. consiste in $|h_n|$ sottrazioni di numeri di circa $\log d$ cifre, cioè ha complessità in tempo $O(d \log d)$.

Infine `memgraph_create` genera il grafo e i nodi dei figli di h_n, copiando i puntatori dei loro codici, cioè ha complessità $\Theta(|h_n|) = O(d)$ sia in tempo che in spazio. La complessità totale quindi è $\Theta(d \log d)$ in tempo, $O(d \log d)$ in spazio.

4.2 Costruzione del grafo

Una volta inizializzato il grafo di appartenenza con i figli di h_n si può costruire l'intero grafo.

```
0  struct dl_list_b *dllb_scanchildren()
1  {
2          struct dl_list_b *l; /* risultato */
3          struct dl_list_b *scan;
4          struct bignat *i; /* contatore */
5          struct bignat *tbn;
6          int c; /* carattere corrente */
7
8          l = dllb_create();
9
10         i = bn_zero();
11         c = getchar();
12         while (!isspace(c) && c != EOF) {
13                 if (c != '0')
14                         l = dllb_add(l, bn_copy(i));
15
16                 tbn = i;
17                 i = bn_succ(i);
18                 bn_destroy(tbn);
19
20                 c = getchar();
21         }
22
23         tbn = i;
24         i = bn_pred(i);
25         bn_destroy(tbn);
26
27         scan = l;
28         while (!dllb_isempty(scan)) {
29                 tbn = dllb_get(scan);
30                 dllb_set(scan, bn_sub(i, dllb_get(scan)));
31                 bn_destroy(tbn);
32
33                 scan = dllb_next(scan);
34         }
35
36         return l;
37 }
```

Frammento 4.1: Implementazione di `dllb_scanchildren`.

```
0  struct memgraph *memgraph_create(struct dl_list_b *l)
1  {
2          struct memgraph *G;
3          struct dl_list_b *scan;
4          struct memnode *x;
5
6          G = malloc(sizeof(struct memgraph));
7          G->V = dlln_create();
8          G->xx = dlln_create();
9          G->rcode = NULL;
10
11         scan = l;
12         while (!dllb_isempty(scan)) {
13                 x = memnode_create(dllb_get(scan));
14                 G->V = dlln_add(G->V, x);
15                 G->xx = dlln_add(G->xx, x);
16
17                 scan = dllb_next(scan);
18         }
19
20         return G;
21 }
```

Frammento 4.2: Implementazione di `memgraph_create`.

4.2.1 Idea

L'algoritmo scorre i nodi del grafo in costruzione, che inizialmente contiene già i figli di h_n ordinati da quello con codice di Ackermann maggiore a quello con il minore e:

1. calcola la lista dei codici di Ackermann dei figli del vertice analizzato;
2. unisce le informazioni di questa lista al grafo, cioè:
   - aggiunge al grafo i vertici corrispondenti ai codici dei figli, se non esistono già;
   - aggiunge gli archi dal vertice analizzato ai figli;
3. analizza il vertice successivo e riparte da 1. fino ad esaurimento.

4.2.2 Implementazione

Nell'implementazione di `memgraph_build`, mostrata nel frammento 4.3:

- `children` è la procedura che, dato un nodo del grafo, ne calcola la lista dei codici dei figli;

```
0  void memgraph_build(struct graph *G)
1  {
2          struct dl_list_n *scan;
3          struct dl_list_b *l;
4          struct node *x;
5
6          scan = G->V;
7          while (!dlln_isempty(scan)) {
8                  x = dlln_get(scan);
9
10                 l = children(x);
11                 G->V = join(G->V, x, l);
12                 dllb_destroy(l);
13
14                 scan = dlln_next(scan);
15         }
16 }
```

Frammento 4.3: Implementazione di `memgraph_build`.

```
0  static struct dl_list_b *children(struct node *v)
1  {
2          struct dl_list_b *l = dllb_create();
3
4          for (uint64_t i = bn_bits(v->code) - 1; i < UINT64_MAX; i--) {
5                  if (bn_get(v->code, i))
6                          l = dllb_add(l, bn_fromuint64(i));
7          }
8
9          return l;
10 }
```

Frammento 4.4: Implementazione di `children`.

- `join` si occupa dell'unire queste informazioni ed aggiungere tutti gli archi uscenti dal nodo analizzato da `children`.

4.2.3 Creazione dei codici dei figli

La procedura `children`, dato $v \in V$, crea e restituisce la lista dei codici degli elementi di v, ordinati in modo decrescente.

Implementazione

L'implementazione di `children` è mostrata nel frammento 4.4.

Accenno di correttezza

Una conseguenza della forma di $\mathbb{N}_A$, espressa nella definizione 1.1.4, e della sua biettività, dimostrata dal teorema 2.2.3, è che $y \in v$ se e solo se il $(\mathbb{N}_A(y)+1)$-esimo bit di $\mathbb{N}_A(v)$ è 1. La procedura sfrutta questa proprietà e, contando all'indietro a partire dal bit più significativo di $\mathbb{N}_A(v)$, crea la lista dei codici decrescenti dei figli.

Complessità

Sia g il numero delle cifre binarie del codice di v. La procedura conta da g a 0, salvando gli indici bit pari a 1, cioè la complessità è $\Theta(g \log g)$ in tempo e $O(g \log g)$ e $\Omega(\log g)$ in spazio.[1]

4.2.4 Unione

La procedura `join`, data la lista V dei vertici del grafo, il vertice x e la lista L dei codici dei suoi figli, aggiunge al grafo i figli se non sono già presenti, e aggiunge ogni figlio alla lista di adiacenza di x.

Idea

Viene sfruttato l'ordinamento sia della lista dei codici dei figli di x che della lista dei nodi V in base al loro codice e, partendo dal fondo sia di L che di V:

- se il codice dell'elemento analizzato di L è uguale al codice del nodo v analizzato di V, allora viene aggiunto v alla lista di adiacenza di x e si scorre sia L che V;
- se il codice dell'elemento di L è maggiore del codice di v, allora significa che il vertice con tale codice non appartiene a V quindi viene creato aggiunto a V dopo v e aggiunto alla lista di adiacenza di x, poi si scorre L;
- altrimenti il codice dell'elemento di L è minore di quello di v, quindi v non è un figlio di x e si scorre L.

[1] Da questa parte dell'implementazione si nota la natura esponenziale dell'appartenenza degli insiemi ereditariamente finiti attraverso $\mathbb{N}_A$ e in un certo senso la sua inefficienza. Le limitazioni sull'input n soddisfano qualunque applicazione pratica ed i codici dei figli di h_n sono rappresentati da `bignat`, ma i codici dei figli dei figli di h_n possono già essere rappresentati da un intero di 64 bit.

Implementazione

L'implementazione di `join` è mostrata nel frammento 4.5.

Accenno di correttezza

La correttezza della procedura verge sul fatto che la lista dei vertici V sia ordinata in modo decrescente rispetto ai codici di Ackermann corrispondenti, e che anche la lista dei codici dei figli di x sia decrescente: la lista viene confrontata con i nodi del grafo di appartenenza, a partire dal fondo.

Complessità

Se l è la cardinalità della lista, la procedura esegue l confronti tra `bignat` creati da `children` e per la biettività di $\mathbb{N}_A$ scandisce nel caso peggiore tanti elementi in coda a V quanti sono i bit del codice di x, cioè la complessità non è peggiore di quella della chiamata precedente a `children`.

4.2.5 Accenno di correttezza

Ad ogni iterazione del ciclo while, vengono trovati i figli del vertice analizzato e gli archi uscenti: `children` e `join` mantengono l'ordine decrescente secondo i codici di Ackermann, che per il teorema 2.3.11 è topologico, ed eventualmente scoprono nuovi vertici del grafo aggiungendoli alla porzione di V di nodi ancora da processare. Perciò all'uscita del ciclo G è il grafo di appartenenza di h_n.

4.2.6 Complessità

Per ogni elemento di $\mathrm{trCl}(h_n)$ vengono scandite le sue cifre binarie dalla procedura `children`: la complessità sia in tempo che in spazio nel caso peggiore, cioè quello in cui $|\mathrm{trCl}(h_n)| = d$, con d il numero di caratteri dati in input, è $\Theta(d \cdot \log d \cdot \log\log d)$.

Invece il caso migliore avviene quando l'input è una tetrazione di 2 e

$$|V| = \mathrm{rank}(h_n) = \urcorner\big((\mathbb{N}_A(h_n)\big) = \urcorner(n),$$
$$|E| = |V| - 1 = \urcorner\big(\mathbb{N}_A(h_n)\big) - 1 = \urcorner(n) - 1,$$

con $\urcorner$ l'inverso della tetrazione di 2, funzione che cresce molto lentamente.

```
0  static struct dl_list_n *join(struct dl_list_n *V, struct memnode *x,
1          struct dl_list_b *l)
2  {
3          /* l lista dei codici dei figli di x (decrescenti) */
4          struct dl_list_n *scanV;
5          struct dl_list_b *scanl;
6          struct memnode *y, *z;
7
8          if (dllb_isempty(l))
9                  return V;
10
11         scanV = dlln_last(V);
12         y = dlln_get(scanV);
13         scanl = dllb_last(l);
14         while (!dllb_isempty(scanl)) {
15                 if (bn_cmp(memnode_get(y), dllb_get(scanl)) == 0) {
16                         /* il vertice esiste gia' */
17                         memnode_addarc(x,y);
18
19                         scanl = dllb_prev(scanl);
20                         scanV = dlln_prev(scanV);
21                         y = dlln_get(scanV);
22                 } else if (bn_cmp(memnode_get(y), dllb_get(scanl)) >= 0) {
23                         /* il vertice non esiste */
24                         z = memnode_create(dllb_get(scanl));
25                         V = dlln_insert(V, z, scanV);
26
27                         memnode_addarc(x,dlln_get(dlln_next(scanV)));
28
29                         scanl = dllb_prev(scanl);
30                 } else {
31                         /* non e' figlio di x */
32                         scanV = dlln_prev(scanV);
33                         y = dlln_get(scanV);
34                 }
35         }
36
37         return V;
38 }
```

Frammento 4.5: Implementazione di `join`.

Capitolo 5

La variante a valori reali

Nello studio del calcolo dei valori di $\mathbb{R}_A$ su insiemi ereditariamente finiti si presentano difficoltà riguardanti errori e approssimazioni. Come modellare l'incertezza su numeri reali? Come approssimare la codifica con un calcolatore utilizzando numeri razionali a precisione arbitraria? Le prossime sezioni tentano di ragionare in modo astratto su degli algoritmi per approssimare l'elevamento a potenza di due con esponente un razionale negativo, per poi implementarli grazie alle procedure fornite dai `bignat` ed i `bigfloat`.

5.1 Considerazioni su errore e precisione

Definizione 5.1.1 (Approssimazione per difetto e per eccesso)**.** Dati $x, \tilde{x} \in \mathbb{R}$, diciamo che $\tilde{x}$ approssima per difetto x se e solo se, per definizione,

$$\tilde{x} = x - \varepsilon_x, \quad \text{con} \quad \varepsilon_x \geq 0;$$

allo stesso modo diciamo che $\tilde{x}$ approssima per eccesso x se e solo se, per definizione,

$$\tilde{x} = x + \varepsilon_x, \quad \text{con} \quad \varepsilon_x \geq 0.$$

Inoltre ci riferiamo a $\pm\varepsilon_x$ come l'errore compiuto nell'approssimazione di x.

Esempio. Dati $u = (u_{n-1} \dots u_1 u_0)_b$ e $v = (v_{m-1} \dots v_1 v_0)_b$, supponiamo di voler trovare un'approssimazione del risultato dell'operazione aritmetica generica op su u e v, in simboli

$$w = (w_{q-1} \dots w_1 w_0, w_{-1} \dots)_b = u \,\mathrm{op}\, v.$$

Fissata la precisione desiderata $p \in \mathbb{N}$, ci viene data la procedura `op_trunc` che restituisce

$$\tilde{w} = (w_{q-1} \dots w_1 w_0, w_{-1} \dots w_{-p})_b = w - \underbrace{\sum_{i<-p} \left(w_i b^i\right)}_{=\varepsilon_w}.$$

cioè un'approssimazione per difetto di w per cui vale che $0 \leq \varepsilon_w < 1 \cdot b^{-p}$. In modo simmetrico la procedura `op_round` approssima w per eccesso in modo da commettere un errore minore di b^{-p}.

Definizione 5.1.2 (Funzioni approssimate)**.** Date le funzioni $f\colon \mathbb{Q} \to \mathbb{R}$ e $\tilde{f}\colon \mathbb{Q} \to \mathbb{Q}$, e dato $p \in \mathbb{N}$, diciamo che $\tilde{f}$ approssima per difetto f con precisione p se e solo se, per definizione,

$$\tilde{f}(x) = f(x) - \varepsilon_f, \qquad 0 \leq \varepsilon_f < b^{-p},$$

mentre diciamo che $\tilde{f}$ approssima per eccesso f con precisione p se e solo se, per definizione,

$$\tilde{f}(x) = f(x) + \varepsilon_f, \qquad 0 \leq \varepsilon_f < b^{-p}.$$

Poiché i nostri scopi sono pratici, di molte funzioni possiamo avere o costruire la funzione approssimata. Alcune implementazioni utili sono:

- `trunc` e `round`, che dati un razionale $u = (u_{n-1} \dots u_1 u_0)_b$ e $p \in \mathbb{N}$, approssimano u (o la funzione di uguaglianza $f(x) = x$) per eccesso o per difetto alla precisione p;
- `div_trunc` e `div_round`, dati $u = (u_{n-1} \dots u_1 u_0)_b$, $v = (v_0)_b$ e la precisione $p \in \mathbb{N}$, restituiscono l'approssimazione di u/v rispettivamente per difetto e per eccesso, con precisione p.

Definizione 5.1.3. Diciamo che x è approssimato dall'intervallo (topologicamente) chiuso $X = [a, b]$, con $a, b \in \mathbb{Q}$, se e solo se, per definizione, $x \in X$.

In [8] X è chiamato un numero intervallare, i valori precisi sono intervalli degeneri e viene notato che gli estremi possono non essere rappresentabili da un numero di macchina: in tal caso si effettua *outward rounding*, cioè l'approssimazione per difetto dell'estremo sinistro con il più grande numero di macchina più piccolo dell'estremo stesso, e l'approssimazione per eccesso dell'estremo destro con il più piccolo numero di macchina più grande di esso.

La definizione di intervalli che contengono i valori esatti semplifica il calcolo e lo studio dell'errore, poiché calcolando operazioni aritmetiche sugli estremi si maneggiano dati non affetti da errore.

5.2 Principi di aritmerica intervallare

Dati $X = [a, b]$ e $Y = [c, d]$, se $\bullet$ denota una delle operazioni di addizione, sottrazione, moltiplicazione e divisione, allora il risultato dell'operazione corrispondente applicata a X e Y è

$$X \bullet Y = \{x \bullet y \mid x \in X, y \in Y\}.$$

Addizione e sottrazione

$$X + Y = [a + c, b + d]$$
$$X - Y = [a - d, b - c]$$

Immagine di funzione continua monotona

$$f(X) = \begin{cases} [f(a), f(b)] & \text{se } f \text{ è non decrescente} \\ [f(b), f(a)] & \text{se } f \text{ è non crescente} \end{cases}$$

5.3 Approssimazione di 2^{-X}

In questa sezione si studia un possibile algoritmo di approssimazione, da parte di un calcolatore, del calcolo della potenza di due con esponente un numero reale negativo approssimato da un intervallo di razionali. Supponiamo inoltre di lavorare con numeri razionali in base b, pari a 2 o una sua potenza, avendo a disposizione semplici operazioni aritmetiche e di approssimazione.

5.3.1 Parte intera e MacLaurin per la parte frazionaria

Sia $X = [x_{\min}, x_{\max}]$ con $x_{\min}, x_{\max} > 0$ il numero intervallare di cui si vuole approssimare il reciproco dell'elevamento a potenza di 2. Per i principi di aritmetica intervallare mostrati nella sezione 5.2, e perché $f(z) = 2^{-z}$ è una funzione continua e decrescente,

$$2^{-X} = 2^{-[x_{\min}, x_{\max}]} = [2^{-x_{\max}}, 2^{-x_{\min}}],$$

quindi un algoritmo che approssimi 2^{-X} deve approssimare $2^{-x_{\max}}$ e $2^{-x_{\min}}$ rispettivamente per difetto e per eccesso, per rispettare l'*outward rounding*. D'ora in poi ci riferiremo a $x_{\max}$ e a $x_{\min}$ con x, sapendo di dover approssimare per difetto in un caso e per difetto nell'altro.

Avendo a disposizione funzioni aritmetiche elementari, si è scelto di approssimare il reciproco dell'elevamento a potenza con il suo polinomio di

MacLaurin di ordine arbitrario $n \in \mathbb{N}$ (si veda [7, pp. 215-230]), che indichiamo con T_n: dati $f(z) = 2^{-z}$ e $n \in \mathbb{N}$, allora $f(z) = T_n(z) + R_n(z)$ con

$$T_n(z) = \sum_{i=0}^{n} \left((-1)^i \cdot \frac{(z \log 2)^i}{(i)!} \right), \qquad R_n(z) = \sum_{i>n} \left((-1)^i \cdot \frac{(z \log 2)^i}{(i)!} \right).$$

Una diretta conseguenza è che, per ogni $z \in \mathbb{R}$, $T_n(z) = f(z) - R_n(z)$, cioè $T_n(z)$ approssima $f(z)$ con errore il resto $R_n(z)$ del polinomio di MacLaurin.

Poiché come centro del polinomio si è scelto lo 0, il valore su cui viene calcolato il polinomio deve essere piccolo e ciò si può ottenere separando prima l'estremo x nella sua parte intera $\lfloor x \rfloor$ e parte frazionaria $\{x\}$. Allora

$$\begin{aligned} 2^{-x} &= 2^{-\lfloor x \rfloor} \cdot 2^{-\{x\}} \\ &= 2^{-\lfloor x \rfloor} \cdot \big(T_n(\{x\}) + R_n(\{x\}) \big) \\ &= 2^{-\lfloor x \rfloor} \cdot T_n(\{x\}) + 2^{-\lfloor x \rfloor} \cdot R_n(\{x\}); \end{aligned} \tag{5.1}$$

così facendo il problema non si complica, perché per ipotesi la base b è un multiplo di 2 e la moltiplicazione per una potenza di 2 con esponente intero è un'operazione semplice da eseguire in tale base.

L'equazione (5.1) è equivalente a

$$2^{-\lfloor x \rfloor} \cdot T_n(\{x\}) = 2^{-x} - 2^{-\lfloor x \rfloor} \cdot R_n(\{x\}), \tag{5.2}$$

cioè $2^{-\lfloor x \rfloor} \cdot T_n(\{x\})$ è un'approssimazione di 2^{-x}. Il segno del resto del polinomio, poiché la serie di Taylor converge alternatamente, dipende dalla parità di n: se n è pari, $R_n(\{x\})$ è negativo ed è un'approssimazione per eccesso, se n è dispari allora è un'approssimazione per difetto.

$T_n(\{x\})$ purtroppo non può essere esattamente calcolato per due principali motivi:

- contiene potenze di $\{x\} \cdot \log 2$, valore che deve essere approssimato se $\{x\}$ è diverso di 0, poiché il logaritmo naturale di 2 è trascendente;
- contiene potenze e divisioni per intero e, anche se le prime potrebbero essere rappresentate non affette da errore a discapito delle prestazioni, il risultato delle seconde può non essere rappresentabile con un numero finito di cifre in base b.

Allora è utile definire T'_n, il polinomio di MacLaurin di ordine $n \in \mathbb{N}$ di $f'(z) = e^{-z}$, per cui vale $f'(z) = T'_n(z) + R'_n(z)$ e

$$T'_n(z) = \sum_{i=1}^{n} \left((-1)^i \cdot \frac{z^i}{(i)!} \right), \qquad R'_n(z) = \sum_{i>n} \left((-1)^i \cdot \frac{z^i}{(i)!} \right),$$

utilizzabile al posto di T_n dopo un opportuno cambio di base, perché per le proprietà delle potenze e dei logaritmi

$$\forall z \geq 0 \qquad 2^{-z} = e^{\log 2^{-z}} = e^{-z \log 2}.$$

Sia quindi $y = \{x\} \log 2$ e sia $\tilde{y}$ la sua approssimazione con precisione $p \in \mathbb{N}$, cioè $\tilde{y} = y + \varepsilon_y$ con $0 \leq |\varepsilon_y| < b^{-p}$: questa approssimazione deve essere effettuata prima del calcolo del polinomio perciò si approssima $\{x_{\max}\}$ per eccesso, con ε_y positivo, e $\{x_{\min}\}$ per difetto, con ε_y negativo. Allora riscrivendo la (5.1)

$$\begin{aligned} 2^{-x} &= 2^{-\lfloor x \rfloor} \cdot 2^{-\{x\}} \\ &= 2^{-\lfloor x \rfloor} \cdot e^{-\{x\} \log 2} = 2^{-\lfloor x \rfloor} \cdot e^{-y} \\ &= 2^{-\lfloor x \rfloor} e^{\varepsilon_y} \cdot e^{-\tilde{y}} \\ &= 2^{-\lfloor x \rfloor} e^{\varepsilon_y} \cdot \left(T'_n(\tilde{y}) + R'_n(\tilde{y})\right) \\ &= 2^{-\lfloor x \rfloor} e^{\varepsilon_y} \cdot T'_n(\tilde{y}) \ + \ 2^{-\lfloor x \rfloor} e^{\varepsilon_y} \cdot R'_n(\tilde{y}), \end{aligned} \tag{5.3}$$

La (5.3) è equivalente a

$$2^{-\lfloor x \rfloor} \cdot T'_n(\tilde{y}) = 2^{-x} + \underbrace{\left(\frac{2^{-x}}{e^{\varepsilon_y}} - 2^{-x}\right)}_{\text{cambio di base}} - \underbrace{2^{-\lfloor x \rfloor} \cdot R'_n(\tilde{y})}_{\text{resto polin.}}, \tag{5.4}$$

cioè $2^{-\lfloor x \rfloor} \cdot T'_n(\tilde{y})$ è un'approssimazione di 2^{-x}. Purtroppo sono ancora presenti le difficoltà del calcolare con esattezza le moltiplicazioni e le divisioni del polinomio.

```
rec_exp_e_frac(ty, n, q)
{
        res = 0;
        for (i = n; i > 0; i--) {
                if (i%2 == 0)
                        res += +1;
                else
                        res += -1;

                res *= ty;
                res = approx(res, q);

                res = div_approx(res, i, q);
        }
        res += 1;

        return res;
}
```

Frammento 5.1: Pseudocodice per `rec_exp_e_frac`.

5.3.2 Approssimare il polinomio di MacLaurin

Come abbiamo notato anche $T'_n(\tilde{y})$ deve essere approssimato: per tale scopo è stata scritta una leggera variante dell'algoritmo di Horner per il calcolo di polinomi (si veda [9, pp. 59-61]), il cui pseudocodice è mostrato nel frammento 5.1. Questa procedura, dato il valore per cui calcolare il polinomio $\tilde{y} \in \mathbb{Q}$ e data la precisione $q \in \mathbb{N}$, utilizza `approx` e `div_approx`, che corrispondono a `trunc` e `div_trunc` se si approssima per difetto, `round` e `div_round` se per eccesso, per stimare il valore di ogni moltiplicazione per $\tilde{y}$ e divisione per intero con precisione q.

Supponiamo di voler stimare $T'_n(\tilde{y})$ per difetto: allora sia n dispari. Lo schema in figura 5.1 mostra il del valore di `res` durante vari momenti dell'esecuzione di `rec_exp_e_frac` e determina che il risultato $\tilde{T}'_n(\tilde{y})$ di tale operazione tale che

$$\tilde{T}'_n(\tilde{y}) = T'_n(\tilde{y}) - \sum_{i=2}^{n} \left(\frac{\tilde{y}^{i-1}}{(i)!} \alpha_i \right) - \sum_{i=2}^{n} \left(\frac{\tilde{y}^{i-2}}{(i-1)!} \beta_i \right),$$

cioè è un'approssimazione per difetto di $T'_n(\tilde{y})$. In modo simmetrico, se volessimo calcolare $T'_n(\tilde{y})$ per eccesso, n dovrebbe essere pari e avremmo che

$$\tilde{T}'_n(\tilde{y}) = T'_n(\tilde{y}) + \sum_{i=2}^{n} \left(\frac{\tilde{y}^{i-1}}{(i)!} \alpha_i \right) + \sum_{i=2}^{n} \left(\frac{\tilde{y}^{i-2}}{(i-1)!} \beta_i \right).$$

`res`	`i`	Prossima operazione
-1	n	`res *= ty`
$-\tilde{y}$	n	`res = trunc(res, q)`
$-\tilde{y} - \alpha_n$	n	`res = div_trunc(res, i, q)`
$-\frac{\tilde{y}}{n} - \frac{\alpha_n}{n} - \beta_n$	$n-1$	`res += 1`
$-\frac{\tilde{y}}{n} + 1 - \frac{\alpha_n}{n} - \beta_n$	$n-1$	`res *= ty`
$-\frac{\tilde{y}^2}{n} + \tilde{y} - \frac{\tilde{y}}{n}\alpha_n - \tilde{y} \cdot \beta_n$	$n-1$	`res = trunc(res, q)`
$-\frac{\tilde{y}^2}{n} + \tilde{y} - \frac{\tilde{y}}{n}\alpha_n - \tilde{y} \cdot \beta_n - \alpha_{n-1}$	$n-1$	`res = div_trunc(res, i, q)`
$-\frac{\tilde{y}^2}{n(n-1)} + \frac{\tilde{y}}{n-1} - \frac{\tilde{y}\alpha_n}{n(n-1)} - \frac{\tilde{y}\beta_n}{n-1} - \frac{\alpha_{n-1}}{n-1} - \beta_{n-1}$	$n-1$	`res = div_trunc(res, i, q)`
$\vdots$		
$\tilde{T}'_n(\tilde{y}) = \underbrace{\sum_{i=0}^{n}\left((-1)^i\frac{\tilde{y}^i}{(i)!}\right)}_{=T(\tilde{y})'} - \sum_{i=2}^{n}\left(\frac{\tilde{y}^{i-1}}{(i)!}\alpha_i\right) - \sum_{i=2}^{n}\left(\frac{\tilde{y}^{i-2}}{(i-1)!}\beta_i\right)$		

Figura 5.1: Calcolo dell'approssimazione per difetto $\tilde{T}'_n(\tilde{y})$; α_i e β_i sono gli errori commessi dall'approssimazione rispettivamente della (i)-esima moltiplicazione per $\tilde{y}$ e della divisione alla $(i-n+1)$-esima iterazione.

Chiamando l'errore ε_T e riprendendo la (5.4) abbiamo che

$$\begin{aligned}(5.4) &\iff 2^{-\lfloor x \rfloor} \cdot \left(\tilde{T}'_n(\tilde{y}) - \varepsilon_T\right) = 2^{-x} + \left(\frac{2^{-x}}{e^{\varepsilon_y}} - 2^{-x}\right) - 2^{-\lfloor x \rfloor} \cdot R'_n(\tilde{y}) \\ &\iff 2^{-\lfloor x \rfloor} \cdot \tilde{T}'_n(\tilde{y}) = 2^{-x} + \underbrace{\left(\frac{2^{-x}}{e^{\varepsilon_y}} - 2^{-x}\right)}_{\text{cambio di base}} - \underbrace{2^{-\lfloor x \rfloor} \cdot R'_n(\tilde{y})}_{\text{resto polin.}} + \underbrace{2^{-\lfloor x \rfloor} \cdot \varepsilon_T}_{\text{appr. polin.}},\end{aligned} \tag{5.5}$$

cioè $2^{-\lfloor x \rfloor} \cdot \tilde{T}'_n(\tilde{y})$ è un'approssimazione di 2^{-x}. Un'utile stima per eccesso (in valore assoluto) di ε_T si ottiene notando che le sommatorie che lo compongono sono parte del polinomio di MacLaurin di ordine n di $f(z) = e^z$ e, poiché ogni operazione è approssimata con precisione q,

$$|\varepsilon_T| < e^{\log 2} \cdot b^{-q} + e^{\log 2} \cdot b^{-q} = 4 \cdot b^{-q}. \tag{5.6}$$

5.3.3 Conclusioni

Ricapitolando, l'approssimazione di 2^{-X} si conduce all'approssimazione degli estremi dell'intervallo $[2^{-x_{\max}}, 2^{-x_{\min}}]$ seguendo il principio di *outward rounding*: trovare un'approssimazione per difetto di $2^{-x_{\max}}$ ed una per eccesso di $2^{-x_{\min}}$. L'algoritmo immaginato esegue le seguenti operazioni:

1. divide $x_{\max}$ o $x_{\min}$ in parte intera e frazionaria, spezzando la potenza in due fattori;
2. della parte frazionaria calcola l'approssimazione della moltiplicazione per $\log 2$ con precisione p (per eccesso per $x_{\max} \log 2$, per difetto per $x_{\min} \log 2$), sostanzialmente cambiando la base della potenza problematica da 2 ad e;
3. approssima il polinomio di MacLaurin in questo punto (di ordine pari per approssimare per difetto, dispari altrimenti) di $f(z) = e^{-z}$ con una leggera variante dell'algoritmo di Horner, approssimando alla precisione q ogni moltiplicazione e divisione;
4. infine moltiplica questo risultato per l'elevamento a potenza di due della parte intera trovato al punto 1. ottenendo l'estremo cercato.

Quindi, dato $X = [x_{\min}, x_{\max}]$ (o $X = (x_{\min}, x_{\max})$), dati $n, m \in \mathbb{N}$ tali che n è dispari e m è pari e dati $p, q \in \mathbb{N}$, l'algoritmo trova gli estremi dell'intervallo

$$\left(2^{-\lfloor x_{\max} \rfloor} \cdot \tilde{T}'_n(\tilde{y}_{\max}),\ 2^{-\lfloor x_{\min} \rfloor} \cdot \tilde{T}'_m(\tilde{y}_{\min})\right), \tag{5.7}$$

che sicuramente è un sovrainsieme di X. Per l'estremo inferiore vale

$$2^{-\lfloor x_{\max} \rfloor} \cdot \tilde{T}'_n(\tilde{y}_{\max}) = 2^{-x_{\max}} + \underbrace{\left(\frac{2^{-x_{\max}}}{e^{\varepsilon_{y_{\max}}}} - 2^{-x_{\max}} \right)}_{\text{cambio di base}} - \underbrace{\frac{R'_n(\tilde{y}_{\max})}{2^{\lfloor x_{\max} \rfloor}}}_{\text{resto polin.}} + \underbrace{\frac{\varepsilon_T^{\max}}{2^{\lfloor x_{\max} \rfloor}}}_{\text{appr. polin.}},$$

con $\tilde{y}_{\max}$ l'approssimazione per eccesso di $\{x_{\max}\} \log 2$ e $\varepsilon_{y_{\max}}$ l'errore (positivo) commesso, $R'_n(\tilde{y})$ il resto del polinomio di MacLaurin di ordine n di $f(z) = e^{-z}$ e $\varepsilon_T^{\max}$ l'errore (negativo) commesso nel calcolo di $\tilde{T}'_n(\tilde{y})$. In modo simmetrico, per l'estremo superiore vale che

$$2^{-\lfloor x_{\min} \rfloor} \cdot \tilde{T}'_m(\tilde{y}_{\min}) = 2^{-x_{\min}} + \underbrace{\left(\frac{2^{-x_{\min}}}{e^{\varepsilon_{y_{\min}}}} - 2^{-x_{\min}} \right)}_{\text{cambio di base}} - \underbrace{\frac{R'_m(\tilde{y}_{\min})}{2^{\lfloor x_{\min} \rfloor}}}_{\text{resto polin.}} + \underbrace{\frac{\varepsilon_T^{\min}}{2^{\lfloor x_{\min} \rfloor}}}_{\text{appr. polin.}},$$

con $\tilde{y}_{\min}$ l'approssimazione per difetto di $\{x_{\min}\} \log 2$ e $\varepsilon_{y_{\min}}$ l'errore (negativo) commesso, $R'_m(\tilde{y})$ il resto del polinomio di MacLaurin di ordine m di $f(z) = e^{-z}$ e $\varepsilon_T^{\min}$ l'errore (positivo) commesso nel calcolo di $\tilde{T}'_m(\tilde{y})$.

Se si volesse stimare l'errore compiuto, delle maggiorazioni utili dell'errore commesso introducendo il polinomio di MacLaurin e nel suo calcolo sono:

- $|R'_n(\tilde{y})| < \dfrac{\tilde{y}^n}{(n)!} \leq \dfrac{(\log 2)^n}{(n)!}$ e $|R'_m(\tilde{y})| < \dfrac{\tilde{y}^m}{(m)!} \leq \dfrac{(\log 2)^m}{(m)!}$, perché la serie di Taylor converge alternatamente e $\tilde{y}$ è l'approssimazione di un valore tra 0 e 1 moltiplicato per $\log 2$;

- $|\varepsilon_T^{\max}| < 4 \cdot b^{-q}$ e $|\varepsilon_T^{\min}| < 4 \cdot b^{-q}$ per la disequazione (5.6), in cui b è la potenza di due che viene utilizzata come base per rappresentare i numeri di macchina a precisione arbitraria.

Purtroppo un modo per stimare l'errore del cambio di base non sembra facilmente ottenibile.

5.4 Implementazione

I concetti discussi nelle sezioni precedenti sono stati implementati nel codice della `struct interval`, usando i `bigfloat`, numeri razionali a precisione arbitraria in base 2^{32}, e le operazioni di calcolo e approssimazione fornite da questi. Le due operazioni importanti su intervalli sono:

- `interval_add`, che calcola l'addizione di due intervalli;

- `interval_rec_exp_2`, che dato l'intervallo X ed i parametri e, m ed a calcola un'approssimazione di 2^{-X} in cui il cambio di base è approssimato con precisione e, il polinomio di MacLaurin ha ordine m o $m+1$ e nel calcolo della sua approssimazione moltiplicazioni e divisioni sono approssimate con precisione a.

Quindi, scelti a priori e, m ed a, per calcolare l'approssimazione di $\mathbb{R}_A(h_n)$ la procedura `graph_calcrack` trova il codice di $\mathbb{R}_A$ per ogni nodo del grafo di appartenenza seguendo l'ordinamento topologico inverso, per poi calcolare il codice di h_n grazie alla lista dei suoi figli. Viene sfruttato il risultato del lemma 2.4.1 e quindi il codice del singoletto di ogni nodo viene calcolato una volta sola.

La complessità in tempo, scelti e ed a vicini, è pari alla complessità di $|V|$ calcoli dei codici di singoletti, e a quella di $|E|$ addizioni: ignorando la complessità dovuta dalla parte intera dei codici in gioco, si tratta di m moltiplicazioni e divisioni su dei razionali tra 0 e 1 con a cifre dopo la virgola, cioè

$$\Theta\Big(|V| \cdot \big(m \cdot a + m \cdot M(a,a)\big) \ + \ |E| \cdot a\Big) = \Theta\big(|V| \cdot ma^2 \ + \ |E| \cdot a\big),$$

in cui $M(a,a)$ è la complessità della moltiplicazione tra due interi di a cifre.

Capitolo 6

Conclusioni

Lo studio del problema e di soluzioni algoritmiche sembra lontano dall'essere conclusivo soprattutto per la seconda parte del problema: mentre è difficile immaginare che un algoritmo per la costruzione del grafo di appartenenza di h_n a partire da $\mathbb{N}_A(h_n)$ non debba esaminare le cifre binarie dei codici di ogni elemento della chiusura transitiva di h_n, lo studio dell'approssimazione di $\mathbb{R}_A$ scalfisce solo la superficie, poiché un buon modo di stimare a priori l'errore commesso sembra di difficile elaborazione, principalmente per la dipendenza dell'errore commesso nel cambio di base rispetto ai dati da calcolare.

Possibili ottimizzazioni e futuri sviluppi sono:

- ottimizzare lo spazio occupato dal grafo di appartenenza, poiché è possibile conoscere un nodo e i suoi discendenti se si conosce il suo codice di Ackermann, oppure si ha la lista dei suoi figli ed essi si possono conoscere (una volta trovati i figli, il codice di Ackermann non è più necessario);
- un migliore algoritmo di moltiplicazione di interi non negativi, come suggerito in [3, p. 294];
- trovare un modo di stimare l'errore commesso nel calcolo di $\mathbb{R}_A$ e determinare i parametri di accuratezza e, n ed a necessari per limitare a piacere l'errore;

mentre una domanda interessante sulla natura dei grafi di appartenenza è: dato $G = (V, E)$ di appartenenza, che relazione c'è tra $|V|$ ed $|E|$?

Infine dei quesiti utili per future applicazioni di $\mathbb{R}_A$, già posti in [1], sono:

- i valori di $\mathbb{R}_A$ per insiemi ereditariamente finiti con rango maggiore di 2 sono trascendenti?

- $\mathbb{R}_A$ è iniettiva? Qual è il minor numero di cifre necessarie a distinguere i codici di due insiemi ereditariamente finiti?

Bibliografia

[1] Eugenio G. Omodeo, Alberto Policriti e Alexandru I. Tomescu. *On Sets and Graphs*. Springer, 2017.

[2] Leonid Libkin. *Elements of Finite Model Theory*. Springer, 2012.

[3] Donald E. Knuth. *The Art of Computer Programming*, Volume 2, Third Edition. Addison-Wesley, 1998.

[4] Brian W. Kernighan e Dennis M. Ritchie. *The C Programming Language*, Second Edition. Prentice Hall, 1988.

[5] IEEE and The Open Group. *<stdint.h>*. The Open Group Base Specification, Issue 7, 2018. `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/stdint.h.html`

[6] IEEE and The Open Group. *<inttypes.h>*. The Open Group Base Specification, Issue 7, 2018. `http://pubs.opengroup.org/onlinepubs/9699919799/basedefs/inttypes.h.html`

[7] Michiel Bertsch, Roberta Dal Passo e Lorenzo Giacomelli. *Analisi matematica*, Seconda Edizione. McGraw-Hill, 2011.

[8] Eldon Hanses e G. William Walster. *Global Optimization Using Interval Analysis*, Second Edition, Revised and Expanded. Marcel Dekker, Inc., 2004.

[9] Alfio Quarteroni e Fausto Saleri. *Introduzione al Calcolo Scientifico*, Terza edizione. Springer, 2006.