Università degli Studi di Udine

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

Relazione del progetto
per il laboratorio dell'insegnamento
di Algoritmi e Strutture Dati

Progettazione e analisi della complessità di un algoritmo per la catena massima di uno *strict poset* di parole di un testo

Studente:
`github.com/nrizzo`

Professore dell'insegnamento:
Alberto Policriti

Anno Accademico 2016–2017 - secondo semestre

Indice

1 Introduzione

1.1 Come compilare il progetto

Il progetto è scritto in C e utilizza un `Makefile` per produrre l'algoritmo risolutivo (chiamato `algoritmo_risolutivo`) e il programma per il calcolo dei tempi (chiamato `calcolo_tempi`). I seguenti comandi, da eseguire nella cartella del progetto, generano i due eseguibili ed eliminano i file temporanei.

```
$ make
$ make clean
```

Inoltre, l'esecuzione di `calcolo_tempi` necessita dei file `corpus2.txt` e `corpus3.txt` nel percorso da cui viene lanciato.

1.2 Struttura del progetto

- `src/main.c` contiene l'algoritmo risolutivo;
- i file in `src/text` si occupano di raccogliere l'alfabeto e le parole del testo da *standard input*;
- i file in `src/text/utf8char` gestiscono i caratteri in codifica UTF-8;
- i file in `src/text/word` gestiscono una parola di caratteri;
- i file in `src/text/alpha` gestiscono l'alfabeto;
- i file in `src/list` implementano le liste;
- i file in `src/graph` e `src/graph/node` implementano il grafo;
- i file in `src/graph/cvector` si occupano di rappresentare i vettori dei caratteri;
- i file in `src/algs` contengono gli algoritmi principali, assieme alle loro procedure ausiliarie (`aux.c` e `aux.h`);
- `src/tcalc/tcalc.c` contiene il codice per il calcolo sperimentale dei tempi di esecuzione dell'algoritmo;
- i file in `src/tcalc/talgs` contengono gli algoritmi utili al calcolo dei tempi incluso il generatore di numeri pseudo-casuali in `src/tcalc/talgs/prng`.
- i file `corpus2.txt` e `corpus3.txt` servono per generare la distribuzione dei casi medi per il calcolo dei tempi;

I programmi non appartenenti all'algoritmo risolutivo o al calcolo dei tempi si trovano in `other`.

1.3 Codifica del testo

Il programma prevede da *standard input* del testo codificato in UTF-8, come è l'impostazione predefinita della maggior parte dei terminali e dei sistemi operativi: ogni carattere definito dalla codifica è riconosciuto e può essere utilizzato.[1]

[1] Sì, anche gli emoji; ma emoji composti da più caratteri (ad esempio con i modificatori) verranno separati.

2 Descrizione del problema

2.1 Esposizione del problema

Definizione 2.1. Sia Σ un alfabeto finito e ordinato (arbitrariamente) tale che il carattere $\flat \in \Sigma$ ($\flat$ è il *blank* o spazio vuoto). Dato $T \in \Sigma^*$, definiamo **parola in** T una qualsiasi stringa $w \in (\Sigma \setminus \{\flat\})^+$ (cioè una stringa non vuota di caratteri diversi da $\flat$) tale che sia una sottostringa di T separata dalle altre sottostringhe da uno o più *blank*.

Definizione 2.2. Dato l'alfabeto finito e ordinato Σ tale che $\flat \in \Sigma$, definiamo S come **l'alfabeto dei caratteri non *blank***, cioè tale che $S = \Sigma \setminus \{\flat\}$ e $|S| = |\Sigma| - 1$.

Quindi, unendo le definizioni precedenti, le parole in un testo $T \in \Sigma^*$ sono tutte le sottostringhe non vuote $w \in S^+$ separate fra di loro da uno o più *blank*.

Definizione 2.3. Data una parola w in un testo $T \in \Sigma^*$, indichiamo con $\vec{w}$ il **vettore dei caratteri di** w, cioè il vettore di lunghezza $|S|$ tale che, per ogni $i \in \{1, \ldots, |S|\}$,

$$\vec{w}[i] = \Big|\{j \mid w[j] \text{ è l'i-esimo carattere nell'ordine di } S\}\Big|.$$

Definizione 2.4. La **relazione binaria** $>$ tra due vettori dei caratteri $\vec{u}$ e $\vec{v}$ è definita nel seguente modo:

$$\vec{u} > \vec{v} \quad \text{sse} \quad \vec{u} \neq \vec{v} \ \wedge \ \forall i \in \{1, \ldots, |S|\}\ \vec{u}[i] \geq \vec{v}[i]$$

Definizione 2.5. Data una parola u in $T \in \Sigma^*$, definiamo $u_\#$ come la sua **lunghezza**. Questa è tale che

$$u_\# = \sum_{j=1}^{|S|} \vec{u}[j].$$

Definizione 2.6. Dato un testo $T \in \Sigma^*$, sia $G_T = (V_T, E_T)$ il grafo orientato definito come segue:

$$V_T = \big\{w \mid w \text{ è una parola in } T\big\},$$

$$E_T = \big\{\langle u, v\rangle \mid u, v \in V_T \wedge \vec{u} > \vec{v}\big\}.$$

Si noti che parole uguali corrispondono ad un unico nodo. Inoltre per l'appartenenza di un arco $\langle u, v\rangle$ a E_T è stata omessa la condizione $u \neq v$ poiché non è necessaria: se $u = v$ allora $\vec{u} = \vec{v}$ e per la definizione 2.4 $\vec{u} \not> \vec{v}$; la contronominale è $\vec{u} > \vec{v}$ implica $u \neq v$.

Problema 2.7. Dato un testo $T \in \Sigma^*$ si calcoli la massima lunghezza di un cammino tra due nodi qualsiasi di G_T.

Specifiche di input e output

L'input è un testo letto da linea di comando (*standard input*). L'output è scritto a terminale (*standard output*) e consiste in un intero che rappresenta il valore cercato, seguito dalla rappresentazione di G_T in formato *dot* in cui i nodi sono etichettati con le parole corrispondenti.

2.2 Considerazioni notevoli

Teorema 2.8. Date u e v parole in T, se $\vec{u} > \vec{v}$ allora $u_\# > v_\#$.

Dimostrazione. Se $\vec{u} > \vec{v}$, per la definizione 2.4 $\forall i \in \{1, \ldots, |S|\}\ \vec{u}[i] \geq \vec{v}[i]$. Perciò, applicando la definizione 2.5,

$$u_\# \ = \ \sum_{j=1}^{|S|} \vec{u}[j] \ \geq \ \sum_{j=1}^{|S|} \vec{v}[j] \ = \ v_\#.$$

Ma $u_\# = v_\#$ solo nei casi in cui:

- $\vec{u} = \vec{v}$, ma allora per la definizione 2.4 $\vec{u} \not> \vec{v}$, che è un assurdo;
- esiste un $i \in \{1, \ldots, |S|\}$ tale che $\vec{u}[i] < \vec{v}[i]$, ma allora per la definizione 2.4 $\vec{u} \not> \vec{v}$, che è un assurdo.

Quindi $u_\# > v_\#$. □

Corollario 2.9. Dati u e v parole in T, se $u_\# \leq v_\#$ allora $\vec{u} \not> \vec{v}$.

Dimostrazione. Per il teorema 2.8 se $\vec{u} > \vec{v}$ allora $u_\# > v_\#$. La contronominale è la tesi. □

Teorema 2.10. La relazione $>$ tra vettori di caratteri è transitiva, asimmetrica e irriflessiva.

Dimostrazione. Transitività: Siano $\vec{u}$,$\vec{v}$,$\vec{w}$ vettori di caratteri tali che $\vec{u} > \vec{v}$ e $\vec{v} > \vec{w}$. Allora, per la definizione 2.4,

$$\vec{u} \neq \vec{v} \neq \vec{w} \quad \wedge \quad \forall i \in \{1, \ldots, |S|\}\ \vec{u}[i] \geq \vec{v}[i] \geq \vec{w}[i].$$

Quindi $\vec{u} > \vec{w}$.

Asimmetria: Sia $\vec{u} > \vec{v}$. Per assurdo, supponiamo che $\vec{v} > \vec{u}$; applicando la definizione 2.4 si ha che $\forall i \in \{1, \ldots, |S|\}\ \vec{u}[i] \geq \vec{v}[i] \wedge \vec{u}[i] \leq \vec{v}[i]$. Cioè i vettori dei caratteri coincidono, ma per la stessa definizione 2.4 non sono in relazione tra di loro: in particolare $\vec{v} \not> \vec{u}$, che è un assurdo. Quindi se $\vec{u} > \vec{v}$ allora $\vec{v} \not> \vec{u}$.

Irriflessività: $\vec{u} \not> \vec{u}$ direttamente dalla definizione 2.4. □

Corollario 2.11. La relazione $>$ tra vettori di caratteri è una relazione di ordine stretto parziale (si veda [2]).

Dimostrazione. Immediata dal teorema 2.10. □

Teorema 2.12. G_T è aciclico.

Dimostrazione. Se, per assurdo, esistesse un ciclo, per la transitività di $>$ dimostrata dal teorema 2.10 tutti i nodi del ciclo (ce n'è almeno uno) sarebbero in relazione con sé stessi: è un assurdo, poiché $>$ è irriflessiva. Quindi G_T è aciclico. □

Definizione 2.13. Date u e v parole in T, diciamo che u è un **anagramma** di v (e viceversa) se e solo se $\vec{u} = \vec{v}$.

Teorema 2.14. Sia $\vec{u} > \vec{v}$ e siano u' un anagramma di u, v' un anagramma di v. Allora $\vec{u'} > \vec{v'}$.

Dimostrazione. Immediata per semplice sostituzione nella definizione 2.4, poiché $\vec{u} = \vec{u'}$ e $\vec{v} = \vec{v'}$. □

G_T ha come vertici le parole di T, ma gli archi sono definiti in base ai vettori dei caratteri. Ai fini dell'algoritmo risolutivo è utile definire un grafo simile a G_T in cui i nodi rappresentano però i **gruppi di anagrammi**, cioè in cui c'è un nodo per vettore dei caratteri.[2]

Definizione 2.15. Dato un testo $T \in \Sigma^*$, sia $G_T^* = (V_T^*, E_T^*)$ il grafo orientato definito come segue:

$$V_T^* = \big\{\vec{w} \mid \vec{w} \text{ è il vettore dei caratteri di una parola in } T\big\}$$

$$E_T^* = \big\{\langle \vec{u}, \vec{v}\rangle \mid \vec{u}, \vec{v} \in V_T^* \wedge\ \vec{u} > \vec{v}\big\}$$

Si noti che vettori dei caratteri uguali corrispondono a un solo nodo.

Si può considerare G_T^* come risultante da G_T dopo aver raggruppato tra di loro gli anagrammi, cioè isomorfo ad un sottografo di G_T in cui viene mantenuto un solo rappresentante per gruppo di anagrammi: anche G_T^* è aciclico. Inoltre questa "trasformazione" può essere annullata, sapendo quali nodi sono stati rimossi: infatti, per il teorema 2.14, se $\langle \vec{u}, \vec{v}\rangle \in E_T^*$ allora per ogni $u' \in V_T$ anagramma di u e per ogni $v' \in V_T$ anagramma di v, $\langle u', v'\rangle \in E_T$.

Il seguente lemma invece verrà usato nelle considerazioni sul calcolo degli archi (sezione 4.7).

Lemma 2.16. Siano u e v parole in $T \in \Sigma^*$ tali che $\vec{u} \neq \vec{v}$. Allora $\vec{u} \not> \vec{v} \ \wedge\ \vec{v} \not> \vec{u}$ (cioè i vettori dei caratteri sono incomparabili) se e solo se esistono $j, k \in \{1, \ldots, |S|\}$ distinti tali che $\vec{u}[j] > \vec{v}[j] \ \wedge\ \vec{u}[k] < \vec{v}[k]$.

Dimostrazione. ($\Rightarrow$) Una diretta conseguenza della definizione 2.4 è

$$\vec{u} \not> \vec{v} \quad \text{sse} \quad \vec{u} = \vec{v} \ \vee\ \neg\big(\forall i \in \{1, \ldots, |S|\}\ \vec{u}[i] \geq \vec{v}[i]\big).$$

Sapendo che $\vec{u} \not> \vec{v}$, allora $\vec{u} = \vec{v} \ \vee\ \neg\big(\forall i \in \{1, \ldots, |S|\}\ \vec{u}[i] \geq \vec{v}[i]\big)$. Ma $\vec{u} \neq \vec{v}$, quindi

$$\exists k \in \{1, \ldots, |S|\}\ \vec{u}[k] < \vec{v}[k].$$

Allo stesso modo, dalla definizione 2.4, da $\vec{v} \not> \vec{u}$ e da $\vec{v} \neq \vec{u}$ si deduce che

$$\exists j \in \{1, \ldots, |S|\}\ \vec{u}[j] > \vec{v}[j].$$

Con ovviamente j e k distinti.

($\Leftarrow$) Sia $j \in \{1, \ldots, |S|\}$ tale che $\vec{u}[j] > \vec{v}[j]$. Supponendo per assurdo che $\vec{v} > \vec{u}$, per la definizione 2.4, $\vec{u}[j] \leq \vec{v}[j]$, che è una contraddizione: allora $\vec{u} \not> \vec{v}$. Allo stesso modo, se esiste $k \in \{1, \ldots, |S|\}$ tale che $\vec{u}[k] < \vec{v}[k]$, allora per la definizione 2.4 $\vec{v} \not> \vec{u}$. □

[2] Nella costruzione del grafo, non è necessario confrontare le parole della stessa lunghezza (quali gli anagrammi) tra di loro, poiché per il corollario 2.9 i loro vettori dei caratteri non possono essere in relazione. Ciò nonostante, un confronto solo tra due elementi di due gruppi di anagrammi di lunghezza diversa determina tutti gli archi coinvolti!

2.3 Ordinamenti topologici

Poiché abbiamo a che fare con un grafo orientato e aciclico, i suoi nodi si possono ordinare in modo che per ogni arco $\langle x, y\rangle \in G_T$ (o G_T^*), x preceda y.

Teorema 2.17. L'ordinamento decrescente dei nodi di G_T (o di G_T^*) secondo la **lunghezza** delle parole corrispondenti (#) è topologico.

Dimostrazione. Siano x e y due nodi di G_T (o di G_T^*) con $\vec{u}$ e $\vec{v}$ i corrispondenti vettori dei caratteri. Se $\vec{u} > \vec{v}$, per il teorema 2.8 allora $u_\# > v_\#$; quindi nell'ordinamento descritto $x \prec y$. □

Teorema 2.18. Fissato $j \in \{1, \ldots, |S|\}$, l'ordinamento decrescente dei nodi di G_T (o di G_T^*) secondo il **numero di occorrenze del j-esimo carattere** nell'ordine di S e, a parità di valori, secondo la lunghezza delle stringhe (#), è topologico.

Dimostrazione. Siano x e y due nodi di G_T (o di G_T^*) con $\vec{u}$ e $\vec{v}$ i corrispondenti vettori dei caratteri. Se $\vec{u} > \vec{v}$, allora per la definizione 2.4 $\vec{u}[j] \geq \vec{v}[j]$, che si può distinguere in $\vec{u}[j] > \vec{v}[j]$ oppure $\vec{u}[j] = \vec{v}[j]$: nel primo caso, $u \prec v$; nel secondo caso viene seguito l'ordinamento topologico descritto dal teorema 2.17. Quindi $x \prec y$. □

Teorema 2.19. Fissati $j, k \in \{1, \ldots, |S|\}$, l'ordinamento decrescente dei nodi di G_T (o di G_T^*) secondo la **minore tra le occorrenze** del j-esimo e del k-esimo carattere nell'ordine di S e, a parità di valori, secondo la lunghezza delle stringhe (#), è topologico.

Dimostrazione. Siano x e y due nodi di G_T (o di G_T^*) con $\vec{u}$ e $\vec{v}$ i corrispondenti vettori dei caratteri. Se $\vec{u} > \vec{v}$ allora per la definizione 2.4 $\vec{u}[j] \geq \vec{v}[j]$ e $\vec{u}[k] \geq \vec{v}[k]$. Di conseguenza,

$$\min(\vec{u}[j], \vec{u}[k]) \geq \min(\vec{v}[j], \vec{v}[k]).$$

Se è vera l'uguaglianza tra i due valori, viene seguito l'ordinamento topologico descritto dal teorema 2.17 in cui $x \prec y$; altrimenti $x \prec y$. Quindi $x \prec y$ □

Teorema 2.20. Fissati $j, k \in \{1, \ldots, |S|\}$, l'ordinamento decrescente dei nodi di G_T (o di G_T^*) secondo la **maggiore tra le due occorrenze** dei caratteri corrispondenti e, a parità di valori, secondo la lunghezza delle stringhe (#), è topologico.

Dimostrazione. In modo simmetrico rispetto alla dimostrazione del teorema 2.19, il massimo tra due numeri che diminuiscono o rimangono costanti non può aumentare. Quindi $u \prec v$. □

3 Strutture dati

Segue una presentazione delle strutture dati più importanti utilizzate.

3.1 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.

3.1.1 Descrizione

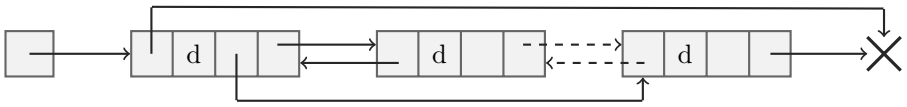


Figura 1: Rappresentazione di una lista.

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.

Nei frammenti di codice mostrati, se l'algoritmo è generico i dati satellite saranno di tipo `data` e gli elementi della lista saranno `struct list`. In pratica le liste in cui i dati sono parole (`struct word`) sono di tipo `struct dl_list_w`, quelle in cui i dati sono nodi (`struct node`) sono di tipo `struct dl_list_n`.

3.1.2 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;
- `get` restituisce il puntatore ai 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 (nei primi due casi, `NULL` se non esiste).

Una scansione di una lista `L` dalla testa può essere effettuata con il seguente frammento di codice.

```
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);
}
```

3.1.3 Complessità

Tutte le operazioni hanno complessità in tempo $\Theta(1)$, poiché contengono un numero finito di *if statement* 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.

3.2 Lista doppiamente concatenata di liste

Per le esigenze dell'algoritmo risolutivo è utile una struttura dati che rappresenti una lista di pile: cioè una lista di liste, entrambe doppiamente concatenate.[3] **Le operazioni** di inserimento e cancellazione **si riferiscono però ai singoli elementi** delle pile. Questa struttura dati è utile per passare da una pila all'altra senza scandire ogni elemento.

3.2.1 Descrizione

Viene seguito lo stesso schema della lista doppiamente concatenata. Una lista di pile è rappresentata dalla pila in testa, `NULL` rappresenta una lista vuota. Inoltre le pile non possono essere vuote.

3.2.2 Operazioni

Le operazioni che differiscono dalla lista doppiamente concatenata sono:

- `add` ora crea e inserisce in coda all'ultima pila un elemento coi dati satellite ricevuti;
- `newpile` crea la pila contenente solo un elemento coi dati satellite ricevuti e la inserisce in coda alla lista di pile;
- `remove`, una volta rimosso l'elemento indicato dalla pila indicata, rimuove anche la pila se è stata svuotata.

[3] Sono tutte liste ma per evitare ambiguità chiamo le liste interne pile (anche nel codice).

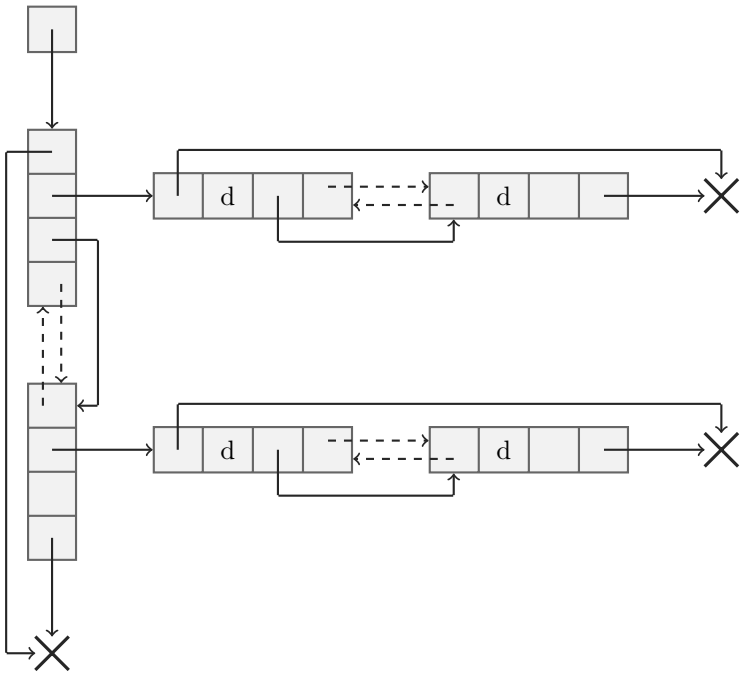


Figura 2: Rappresentazione di una lista (verticale) di pile (orizzontali).

3.2.3 Complessità

Come per la lista doppiamente concatenata, ogni operazione esegue un numero finito di istruzioni, quindi p operazioni vengono eseguite in tempo $\Theta(p)$. Allo stesso modo, gestire una lista di pile con q elementi complessivi richiede spazio $\Theta(q)$, indipendentemente dal numero di pile.

3.3 Grafo orientato: lista di adiacenze

Per rappresentare il grafo G_T (o G_T^*) è stata scelta la struttura delle liste di adiacenza. Più in particolare, la `struct graph` contiene la lista dei vertici del grafo (`struct dl_list_n`), mentre ogni `struct node` ha la lista dei propri figli (sempre `struct dl_list_n`).

3.3.1 Complessità

La creazione di un grafo e l'aggiunta di tutti i suoi archi richiede l'esecuzione di circa $|V|+|E|$ operazioni che costano $\Theta(1)$, quindi la complessità è $\Theta(|V|+|E|) = O(|V|^2)$ sia in tempo che in spazio.

4 Algoritmo risolutivo

Le procedure principali dell'algoritmo risolutivo, in `src/main.c`, sono le seguenti:

- `text_retrieve` legge lo *standard input* carattere per carattere e costruisce l'alfabeto $S = \Sigma \setminus \{\text{b}\}$ e la lista delle parole di T;
- `alg_removeduplicates` prende le parole di T e ne crea una lista senza doppioni;
- `graph_retrievenodes` da questa lista genera i vertici V_T del grafo G_T (senza archi, E_T rimane vuoto);
- `graph_joinanagrams` trasforma V_T in V_T^*, raggruppando (in modo reversibile) gli anagrammi;
- `graph_topologicalsort` dispone V_T^* in un ordinamento topologico;
- `graph_findedges` trova tutti gli archi di G_T^*, riempiendo E_T^*;
- `graph_findllp` calcola per ogni vertice $x \in V_T^*$ la massima lunghezza dei percorsi di G_T^* con primo vertice x e salva il massimo di questi (è la lunghezza del percorso più lungo anche in G_T);
- `graph_printllp` stampa su *standard output* la lunghezza trovata da `graph_findllp`;
- `graph_printDOT` stampa G_T in formato DOT, ricostruendolo da G_T^* grazie agli anagrammi salvati.

4.1 Ordinamento in tempo lineare: `linearsort`

Questo algoritmo di **ordinamento decrescente** viene usato molte volte all'interno del progetto per ordinare una lista di a elementi con chiave compresa nell'intervallo $[0, b]$ (o $[1, b]$).[4]

4.1.1 Idea

L'algoritmo è una variante degli ordinamenti in tempo lineare visti in classe quali BucketSort e CountingSort (si veda [1, pp. 159-160, 164-165]) e ordina una lista. Consiste in:

1. scandire la lista degli elementi da ordinare, mettendoli nei secchielli corrispondenti alla loro chiave;
2. scandire i secchielli in ordine decrescente e comporre così una lista ordinata.

[4] D'ora in avanti, a e b indicheranno il numero di elementi e la chiave massima di `linearsort` o di una sua implementazione.

4.1.2 Implementazione

```
0  struct list *linearsort(struct list *L)
1  {
2          int max; /* chiave massima */
3          int c; /* chiave corrente */
4          struct list *result; /* lista da restituire */
5          struct list *scan; /* elemento corrente */
6          struct list **buckets; /* secchielli per ogni chiave */
7
8          /* Ricerca della chiave massima */
9          max = 0;
10         scan = L;
11         while (!isempty(scan)) {
12                 c = getkey(get(scan));
13
14                 if (c > max)
15                         max = c;
16
17                 scan = next(scan);
18         }
19
20         /* Inizializzazione dei secchielli */
21         buckets = malloc(sizeof(struct list)*(max+1));
22         for (int i=0; i <= max; i++)
23                 buckets[i] = create();
24
25         /* Immissione degli elementi nei secchielli */
26         scan = L;
27         while (!isempty(scan)) {
28                 c = getkey(get(scan));
29
30                 buckets[c] = add(buckets[c],get(scan));
31
32                 scan = next(scan);
33         }
34
35         /* Creazione della lista ordinata e lettura dai secchielli */
36         result = create();
37         for (int i=max; i >= 0; i--) {
38                 scan = buckets[i];
39
40                 while (!isempty(scan)) {
41                         result = add(result,get(scan));
42                         scan = next(scan);
43                 }
44
45                 destroy(buckets[i]);
46         }
47
48         destroy(L);
49         free(buckets);
50         return result;
51 }
```

4.1.3 Accenno di correttezza

Segue qualche considerazione sulla correttezza di `linearsort`:

- l'esecuzione di 10-18 assegna a `max` la chiave massima b, infatti scandisce tutta la lista data aggiornando `max`;
- l'esecuzione di 27-33 mette i nodi nei secchielli corrispondenti alla loro chiave;
- 36-45 ricostruisce la lista ordinata, infatti scandisce i secchielli in ordine decrescente.

4.1.4 Complessità

Sia a la cardinalità della lista da ordinare e b il massimo valore della lista. Allora, come mostrato nella figura 3, la complessità in tempo risulta $\Theta(a + b)$.

Linee	Operazione	Complessità in tempo
10-19	scandire la lista	$\Theta(a)$
22-25	inizializzare b secchielli	$\Theta(b)$
28-35	disporre gli a elementi nei secchielli	$\Theta(a)$
38-48	scandire i secchielli	$\Theta(a + b)$
Totale		$\Theta(a + b)$

Figura 3: Operazioni e loro complessità in tempo.

Oltre alla lista in input, `linearsort` tiene in memoria l'equivalente di due copie della lista (nei secchielli e nel risultato) e deve gestire ogni secchiello, quindi anche la complessità in spazio è $\Theta(a + b)$.

4.1.5 Stabilità

L'algoritmo è stabile, poiché gli elementi vengono aggiunti in coda alle liste dei secchielli e queste vengono lette dalla testa alla coda: se due elementi hanno la stessa chiave la precedenza viene conservata.

4.2 Ottenimento di parole e alfabeto

`text_retrieve` legge il testo T da *standard input* un carattere alla volta e:

- lo aggiunge all'alfabeto $S = \Sigma \setminus \{\text{b}\}$, se non è il carattere *blank* e non è già presente;
- costruisce il vettore delle parole di T, anche assegnando ad ogni parola la sua lunghezza.

4.2.1 Complessità

Costruzione dell'alfabeto L'alfabeto S è rappresentato da un vettore, che può essere ingrandito per far spazio a nuovi caratteri (il codice si trova in `src/text/alpha`). L'inserimento di un carattere mai incontrato prima richiede la scansione di tutto il vettore, per verificare se è già presente o meno, e una chiamata a `realloc` per ingrandire l'array dinamico contenente l'alfabeto. Quindi una stima per eccesso della complessità in tempo è $\Theta(|T| \cdot |S| + |S|^2)$, una stima per difetto $\Theta(|T| \cdot 1)$. La complessità in tempo risulta $\Theta(|T|)$, poiché $|S|$ è costante, mentre la complessità in spazio è $\Theta(|S|) = \Theta(1)$.

Costruzione delle parole `text_retrieve` per ogni carattere letto:

- se è *blank* e se c'è una parola in costruzione la salva nella lista di parole di T;
- altrimenti, se non è *blank*:
  - se c'è una parola in costruzione aggiunge il carattere in coda alla parola;
  - altrimenti inizia a costruire una nuova parola contenente quel carattere.

L'algoritmo esegue questi passaggi fino ad esaurimento del testo, ricordandosi dell'ultima parola in costruzione prima dell'*end-of-file*. Più in dettaglio, per salvare ogni parola utilizza un buffer la cui dimensione viene raddoppiata con `realloc` quando pieno; la complessità di questa procedura dipende dall'implementazione, ma è ragionevole assumere che sia lineare, sia in tempo che in spazio, rispetto alla memoria richiesta.

Non considerando i `realloc`, ad ogni lettura di un carattere l'algoritmo esegue un numero finito di istruzioni. Invece, prendendo in esame solo questi, per ogni parola u letta da *standard input*, l'algoritmo rialloca la parola in costruzione circa $g \in \mathbb{N}$ volte, con 2^g la più piccola potenza di due maggiore di $u_\#$: essa è tale che $2^{g-1} \leq u_\# \leq 2^g$. La complessità totale in tempo delle chiamate di `realloc` per u è quindi

$$\begin{aligned} c_0 + 2c_1 + 2^2c_2 + \cdots + 2^g c_g \;&\leq\; c_{\max} \cdot \sum_{i=0}^{g} 2^i \;=\; c_{\max} \cdot (2^{g+1} - 1) \;\in\; \Theta(2^g) \\ &\geq\; c_{\min} \cdot \sum_{i=0}^{g} 2^i \;=\; c_{\min} \cdot (2^{g+1} - 1) \;\in\; \Theta(2^g), \end{aligned}$$

con c_m per $m \in \{0, \ldots, g\}$ le costanti delle singole chiamate e con $c_{\max} = \max(c_0, \ldots, c_g)$ e $c_{\min} = \min(x_0, \ldots, c_g)$. Poiché

$$2^{g-1} \leq u_\# \leq 2^g \iff 2^g \leq 2u_\# \leq 2^{g+1},$$

la complessità è $\Theta(u_\#)$.

In conclusione, notando che la somma delle lunghezze delle parole deve essere minore o uguale a $|T|$, la complessità totale in tempo della costruzione delle parole è $\Theta(|T|)$. La complessità in spazio invece è $O(|T|)$, poiché T potrebbe anche contenere solo *blank*.

4.3 Eliminazione dei doppioni

Per eliminare i doppioni le parole vengono prima divise secondo la loro lunghezza e poi ogni gruppo di parole viene ordinato lessicograficamente.[5] I doppioni risultano adiacenti, quindi basta una scansione di ogni gruppo per individuarli ed eliminarli.

4.3.1 Implementazione

`src/algs/algs.c`

```
0  struct dl_list_w *alg_removeduplicates(struct text *T)
1  {
```

⋮

```
28         /* Creazione della lista ordinata e lettura dai secchielli */
29         result = dllw_create();
30         for (int i=max; i >= 1; i--) {
31            buckets[i] = lexicographicalsort(T,buckets[i],i);
32            buckets[i] = uniq(buckets[i]);
33
34            scan = buckets[i];
35            while (!dllw_isempty(scan)) {
36                result = dllw_add(result,dllw_get(scan));
37                scan = dllw_next(scan);
38            }
39
40            dllw_destroy(buckets[i]);
41         }
42
43         free(buckets);
44         return result;
45 }
```

Nel codice precedente:

- è stata omessa la porzione di algoritmo che segue la preparazione dei secchielli di `linearsort` secondo la lunghezza delle parole;
- `lexicographicalsort` è un'implementazione di RadixSort (si veda [1, pp. 162-164]) che ordina una lista di parole della stessa lunghezza; utilizza come algoritmo stabile un'implementazione della variante di `linearsort` per l'ordinamento crescente e con chiave l'indice dei caratteri in S;
- `uniq` scandisce la lista ed elimina i doppioni adiacenti.

[5] L'alfabeto $S = \Sigma \setminus \{\mathfrak{b}\}$ è ordinato secondo quale carattere è apparso per la prima volta nel testo.

4.3.2 Accenno di correttezza

La variante di `linearsort` usata in `lexicographicalsort` è stabile, quindi ogni secchiello viene ordinato secondo le parole. I doppioni vengono posti nello stesso secchiello, perché di uguale lunghezza, sono adiacenti e vengono eliminati da `uniq`.

4.3.3 Complessità

Prima di calcolare la complessità è utile sapere la lunghezza massima e il numero massimo delle parole in T.

Teorema 4.1. Dato un testo $T \in \Sigma^*$, la lunghezza di ogni parola in T è $O(|T|)$.

Dimostrazione. Per la definizione 2.1, ogni parola w in T è una sottostringa di T. Quindi $w_\# \leq |T|$ e $w_\# \in O(|T|)$. □

Teorema 4.2. Dato un testo $T \in \Sigma^*$, il numero di parole in T (doppioni inclusi) è $O(|T|)$.

Dimostrazione. Sia n il numero di parole in T. Esse sono sottostringhe di T non intersecanti, sono composte da almeno un carattere e precedono almeno uno spazio[6] (eccetto per l'ultima parola), quindi

$$|T| \geq 2n - 1 \quad \iff \quad n \leq \frac{|T| + 1}{2}.$$

Di conseguenza $n \in O(|T|)$. □

L'eliminazione dei doppioni si scompone in:

1. ordinamento delle parole secondo la loro lunghezza, che consiste in un'implementazione di `linearsort` in cui a è il numero di parole, b la lunghezza massima;

2. esecuzione di `lexicographicalsort`, che per ogni gruppo di parole di una data lunghezza esegue altrettanti `linearsort` con chiave l'indice dei caratteri in S, cioè in cui a è un sottoinsieme delle parole in T, b è $|S|$;

3. esecuzione di `uniq`, che (complessivamente) scandisce una sola volta la lista ordinata delle parole, confrontando ogni parola con la successiva (il confronto, eseguito da `word_equals`, nel caso peggiore è lineare rispetto alla lunghezza della parola).

Di conseguenza, dati:

- S l'alfabeto $\Sigma \setminus \{\mathfrak{b}\}$ di cardinalità costante;
- n il numero di parole in $T \in \Sigma^*$;
- $l_{\max}$ la lunghezza della parola più lunga;

[6] Un esempio più interessante del caso limite si presenta se le parole hanno lunghezza maggiore di 1, poiché il numero delle diverse parole ottenibili cresce esponenzialmente rispetto alla lunghezza fissata. In tale situazione ci possono comunque essere $\Theta(|T|)$ parole.

- $m_1, m_2, \ldots, m_k$ le cardinalità del partizionamento delle parole in T in gruppi di lunghezza $l_1, l_2, \ldots, l_k$;

ricordando i risultati dei teoremi 4.1 e 4.2 e notando che $\sum_{i=1}^{k} m_i l_i \leq |T|$, la complessità in tempo è

$$\overbrace{\Theta(n + l_{\max})}^{1.} + \overbrace{\sum_{i=1}^{k} \Theta(l_i \cdot (m_i + |S|))}^{2.} + \overbrace{O(|T|)}^{3.} = O(|T|),$$

mentre la complessità in spazio corrisponde alla somma dei primi due addendi della formula precedente (`uniq` opera in-place), cioè $O(|T|)$.

4.4 Generazione di V_T e dei vettori dei caratteri

`graph_retrievenodes` legge la lista delle parole senza doppioni e genera V_T, assieme ai vettori dei caratteri di ogni parola.

4.4.1 Complessità

La lista delle parole e le parole stesse vengono lette una volta sola per la creazione dei nodi e dei vettori dei caratteri; per ogni parola e per ogni suo carattere è eseguito un numero finito di istruzioni. Dato n il numero di parole in T, ricordando il risultato del teorema 4.2 e notando che $|V_T| \leq n$, la complessità sia in tempo che in spazio è $\Theta(|V_T|) + O(|T|) = O(|T|)$.

4.5 Da V_T a V_T^*: unione degli anagrammi

I nodi di G_T vengono ordinati secondo loro vettori dei caratteri con un'implementazione di RadixSort. Di conseguenza gli anagrammi risulteranno adiacenti e sarà necessaria una sola scansione per tenere un solo nodo rappresentante per ogni gruppo di anagrammi, aggiungendo le parole scomparse alla sua lista di parole `anagrams` (`struct dl_list_w`).

4.5.1 Implementazione

`src/algs/algs.c`

```
0  struct dl_list_n *alg_joinanagrams(struct alphabet *S, struct dl_list_n *V)
1  {
2          int s = alpha_size(S);
3          struct dl_list_n *prev;
4
5          for (int i = s-1; i >= 0; i--) {
6                  prev = V;
7                  V = sortbycvector(S,V,i);
8                  dlln_destroy(prev);
9          }
10
11         V = uniqcv(V);
12
13         return V;
14 }
```

`sortbycvector` è un'implementazione della variante crescente di `linearsort` che ordina una lista di nodi secondo il loro `i`+1-esimo campo dei vettori dei caratteri (quindi a è $|V_T|$, b è la massima occorrenza dell'`i`+1-esimo carattere in una singola parola); `uniqcv` scandisce la lista dei nodi raggruppando gli anagrammi adiacenti.

4.5.2 Accenno di correttezza

La variante di `linearsort` per l'ordinamento crescente è stabile, quindi i vettori dei caratteri uguali saranno adiacenti; `uniqcv` confronta ogni nodo con il successivo.

4.5.3 Complessità

Teorema 4.3. Il numero massimo di occorrenze di un carattere in una singola parola è $O(|T|)$.

Dimostrazione. Il numero massimo di occorrenze di un qualsiasi carattere di una parola deve essere minore o uguale alla lunghezza della parola stessa. Essa, per il teorema 4.1, è $O(|T|)$. □

Quindi, dati:

- $S = \Sigma \setminus \{\text{ƀ}\}$ di cardinalità costante;
- n il numero di parole in T;
- $j_{\max}$ la massima frequenza di un carattere (qualsiasi) in una parola;

ricordando i risultati dei teoremi 4.2 e 4.3 e notando che $|V_T| \leq n$, la complessità in tempo è

$$O\big(|S| \cdot (|V_T| + j_{\max})\big) + \Theta(|V_T|) = O(|T|).$$

La complessità in spazio invece è pari al primo addendo della formula precedente (`uniqcv` opera in-place), cioè $O(|T|)$.

4.6 Ordinamento topologico

È stato implementato `linearsort` con chiave la lunghezza delle parole, cioè in cui a è $|V_T^*|$ e b è la lunghezza massima di una parola in T.

4.6.1 Correttezza

Per il teorema 2.17, l'ordinamento secondo la lunghezza delle parole è topologico in G_T^*.

4.6.2 Complessità

Dati n il numero di parole in T e l_{max} la lunghezza della parola più lunga in T, ricordando i risultati dei teoremi 4.1 e 4.2 e notando che $|V_T^*| \leq |V_T| \leq n$, la complessità sia in tempo che in spazio risulta

$$\Theta(|V_T^*| + l_{\max}) = O(|T|).$$

4.7 Popolamento di E_T^*

La procedura `findedges` ha il compito di trovare gli archi di G_T^* e utilizza gli ordinamenti topologici descritti nei teoremi 2.17, 2.18, 2.19 e 2.20.

4.7.1 Idea

Un ordinamento topologico è tale che, preso un suo elemento x, i genitori lo precedono nell'ordinamento e i figli lo succedono. L'idea principale dell'algoritmo per il calcolo degli archi è di **tenere in memoria più ordinamenti topologici** in liste dinamiche e di:

1. scegliere il nodo con le due liste più convenienti (che richiedono meno confronti) per il calcolo di genitori e figli;
2. trovare gli archi uscenti ed entranti da esso grazie alle due liste scelte;
3. **rimuoverlo** da tutte le liste;
4. ripetere da 1. fino a esaurimento dei nodi.

#:	rasserenava	serena	nonna	nonno	non	era	se	si
a:	rasserenava	serena	nonna	era	nonno	non	se	si
e:	rasserenava	serena	era	se	nonna	nonno	non	si
i:	si	rasserenava	serena	nonna	nonno	non	era	se

Figura 4: Rappresentazione di qualche ordinamento topologico. # indica l'ordinamento del teorema 2.17, gli altri tre l'ordinamento del teorema 2.18 fissato il carattere corrispondente.

Ad esempio, nella figura 4 il nodo corrispondente alla parola "si" non ha figli né genitori in G_T (o in G_T^*) e può essere immediatamente tolto dalle liste, poiché "si" è ultima nel primo ordinamento e prima nell'ultimo.

Un'ulteriore accorgimento riguarda la **lunghezza delle parole**: una conseguenza del teorema 2.8 è che in un gruppo di parole della stessa lunghezza non ci sono archi[7]. Cioè le parole della stessa lunghezza che si ritrovano adiacenti negli ordinamenti possono essere raggruppate in pile: così nel calcolo degli archi di un nodo x potranno essere ignorati i nodi appartenenti alla sua stessa pila nella lista scelta.

Ad esempio, nella figura 5 il conteggio dei possibili figli e genitori della parola "nonno" può escludere "nonna" (e viceversa) in tutti gli ordinamenti tranne il secondo.

L'algoritmo **immaginato** ha la struttura di un algoritmo goloso e ad ogni iterazione sceglie il nodo più conveniente e lo processa, fino ad esaurimento. Ma questa scelta quanta computazione può richiedere, con l'ulteriore complicazione del raggruppamento in pile di parole di uguale lunghezza?

Si è deciso di implementare un'**approssimazione** di questo algoritmo, la cui bontà non viene mostrata, **che ordina solo inizialmente i nodi** in una coda in base al numero di confronti necessari per trovare figli e genitori per poi elaborarli in quell'ordine, ignorando i cambiamenti risultanti dalla rimozione dei nodi dalle liste.

[7] Insiemi per cui vale questa proprietà sono detti anticatene (si veda [2]).

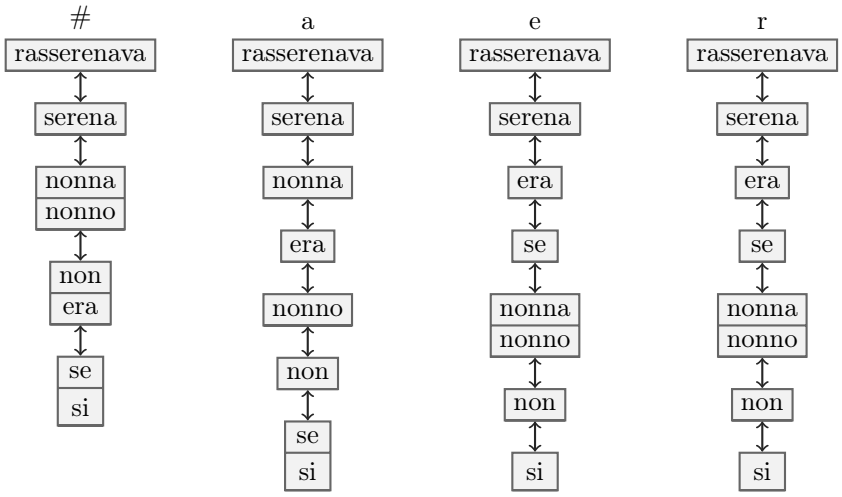


Figura 5: Rappresentazione di alcuni ordinamenti di pile. # indica l'ordinamento del teorema 2.17, gli altri tre l'ordinamento del teorema 2.18 fissato il carattere corrispondente.

4.7.2 Quali ordinamenti scegliere

Un altro problema riguarda quali ordinamenti topologici scegliere tra quelli dimostrati nella sezione 2.3. Si propongono due possibili schemi, anche se la versione finale è stato preferito il primo.

Versione standard Vengono utilizzati questi $|S| + 1 \in \Theta(1)$ ordinamenti:

- l'ordinamento descritto dal teorema 2.17, cioè quello secondo la lunghezza delle parole;
- tutti i diversi ordinamenti possibili descritti dal teorema 2.18, cioè uno per ogni carattere di $|S|$.

Teorema 4.4. Dati x e y vertici distinti di V_T^*, siano $\vec{u}$ e $\vec{v}$ i vettori dei caratteri corrispondenti. Essi sono incomparabili ($\vec{u} \not\succ \vec{v}$ e $\vec{v} \not\succ \vec{u}$) se e solo se tra gli ordinamenti topologici usati nella versione standard esiste un ordinamento in cui $x \prec y$ e uno in cui $y \prec x$.

Dimostrazione. Poiché x e y sono vertici distinti di V_T^*, deve essere che $\vec{u} \neq \vec{v}$ (altrimenti sarebbero lo stesso vertice). Ma allora per il lemma 2.16 sono incomparabili se e solo se esistono $j, k \in \{1, \ldots, |S|\}$ distinti tali che $\vec{u}[j] > \vec{v}[j] \wedge \vec{u}[k] < \vec{v}[k]$. Il successivo della doppia implicazione è equivalente a $x \prec y$ nell'ordinamento descritto dal teorema 2.18 fissato j e a $y \prec x$ nell'ordinamento descritto dallo stesso teorema fissato k, ed entrambi gli ordinamenti sono presenti nello schema della versione standard. □

Intuitivamente, questo schema di ordinamenti è efficace in presenza di disparità nella distribuzione dei caratteri per ogni parola. Però, come visto dai risultati dal calcolo dei tempi per il caso medio della sezione 5.2, sembrerebbe non essere efficace se queste disparità sono distribuite in modo più o meno omogeneo.

Variante potenziata Il tentativo di creare una combinazione di ordinamenti più potente, in particolare per il caso medio della sezione 5.2, ha portato alla progettazione di uno schema di ordinamenti più grande, contenente:

- gli $|S| + 1$ ordinamenti della versione standard;
- per ogni coppia non ordinata $i, j \in \{1, \ldots, |S|\}$ tale che $i \neq j$, l'ordinamento basato sul minore tra i caratteri corrispondenti (teorema 2.19) e quello basato sul maggiore tra i due (teorema 2.20).

Se $|S|$ è costante allora questi ordinamenti saranno $\Theta(|S|^2) = \Theta(1)$. Purtroppo, nella pratica, generare e tenere in memoria $|S|^2$ liste di $|V_T^*|$ elementi può essere **molto costoso**, quindi per l'algoritmo risolutivo finale è stata preferita la versione standard.

Intuitivamente, questo schema dovrebbe risparmiare qualche confronto inutile tra vettori di caratteri che sarebbe avvenuto con solo gli ordinamenti della versione standard.

Ad esempio, dati x e y vertici di G_T^* con vettori dei caratteri $\vec{u}$ e $\vec{v}$:

- sia x il nodo selezionato come più conveniente;
- sia l'ordinamento secondo il carattere j-esimo quello indicato come migliore per il calcolo dei figli di x;
- sia $u_\# > v_\#$;
- sia $\vec{u}[j] \geq \vec{v}[j]$;

se il confronto per determinare se $\langle x, y \rangle \in E_T^*$ fallisce, i vettori dei caratteri sono incomparabili e per il teorema 4.4 esiste l'ordinamento secondo il k-esimo carattere in cui $y \prec x$. Se fosse esistito l'ordinamento secondo il minore tra il j-esimo e il k-esimo carattere, l'algoritmo goloso dovrebbe aver scelto quello, in cui $x \prec y$, ed essersi risparmiato il confronto. Allo stesso modo, la presenza degli ordinamenti secondo la maggiore tra le occorrenze di ogni coppia di caratteri aiuterebbe il calcolo dei genitori.

In questo ragionamento però non sono stati presi in considerazione l'approssimazione della scelta dell'algoritmo goloso e il raggruppamento in pile secondo la lunghezza delle parole corrispondenti ai nodi.

4.7.3 Implementazione

src/algs/algs.c

```
0  void alg_findedges(struct graph *G_T)
1  {
2          struct alphabet *S = G_T->S; /* alfabeto S=Sigma\{blank} */
3          struct tsort_list *tops; /* gli ordinamenti topologici */
4          struct dl_list_n *sorted; /* lista in costruzione */
5          struct node *x; /* nodo corrente */
6          struct tsort_list *tscan; /* lista di ordinamenti topologici */
7          struct dl_list_n *queue; /* coda di elaborazione dei nodi */
8
9          /* Costruzione degli ordinamenti */
10         tops = tsl_create();
11         for (int i=0; i < alpha_size(S); i++) {
12                 tops = tsl_add(tops,top_char(G_T,i));
13         }
14
15         if (G_T->V != NULL)
16                 tops = tsl_add(tops,pileup(G_T->V));
17
18         /* Indicizzazione dei puntatori in tops per ogni nodo */
19         findpointers(G_T->V,tops);
20
21         /* Calcolo di minimi figli e minimi genitori per ogni nodo e ogni
22          *  ordinamento topologico in tops */
23         tscan = tops;
24         while (!tsl_isempty(tscan)) {
25                 calcminchildren(tsl_get(tscan));
26                 calcminparents(tsl_get(tscan));
27                 tscan = tsl_next(tscan);
28         }
29
30         /* Creazione della coda secondo cui processare i nodi */
31         queue = queueup(G_T->V);
32
33         /* Computazione degli archi */
34         while (!dlln_isempty(queue)) {
35                 x = dlln_get(queue);
36
37                 calcchildren(x);
38                 calcparents(x);
39                 topsremove(tops,x);
40
41                 queue = dlln_remove(queue,queue);
42         }
43
44         tsl_destroy(tops);
45         dlln_destroy(queue);
46 }
```

4.7.4 Accenno di correttezza

Le operazioni delle righe 9-19 generano gli ordinamenti impilati dei nodi di V_T^* e indicizzano i loro puntatori in essi per la successiva rimozione, mentre le righe 21-28 scelgono per ogni nodo le due liste migliori (al momento) per l'elaborazione. Indipendentemente dalle liste scelte, esse sono ordinamenti topologici: quando un nodo viene estratto, nella ricerca dei figli viene confrontato con tutti i nodi alla sua destra nell'ordinamento, ad eccezione dei nodi corrispondenti a parole della stessa lunghezza; allo stesso modo, nella ricerca dei genitori viene confrontato con quelli alla sua sinistra. Non vengono dimenticati archi poiché un nodo viene rimosso dalle liste solo dopo aver trovato tutti gli archi entranti e uscenti.

In particolare il confronto tra due nodi `x` e `y`, con u e v le loro rispettive parole in T, viene effettuato dalle procedure `calcchildren` e `calcparents` (il cui codice si trova in `src/algs/aux.c`) tramite l'espressione

```
(xlength > ylength && cv_rel(xcv,ycv))
```

che è vera se e solo se $u_\# > v_\#$ (quindi $\vec{u} \neq \vec{v}$) e $\forall i \in \{1, \ldots, |S|\}$ $\vec{u}[i] \geq \vec{v}[i]$, cioè $\vec{u} > \vec{v}$ per la definizione 2.4.

4.7.5 Complessità

Dato $S = \Sigma \setminus \{\mathfrak{b}\}$ di cardinalità costante, n il numero di parole in T, $j_{\max}$ il massimo numero di occorrenze di un carattere in una parola in T e sapendo che:

- `topchar` implementa `linearsort` secondo il numero di occorrenze del carattere fissato, cioè in cui a è $|V_T^*|$ e b è $j_{\max}$, per poi raggruppare i nodi adiacenti e corrispondenti a parole adiacenti della stessa lunghezza con `pileup`;
- `findpointers` indicizza i puntatori alle liste dinamiche nella `struct` di ogni nodo, scandendo una sola volta le liste;
- anche `calcminchildren` e `calcminparents` complessivamente scandiscono una volta sola le liste, assegnando a ogni nodo la migliore lista per il calcolo di figli e quella per il calcolo dei genitori;
- `queueup` è un'implementazione della variante crescente di `linearsort` secondo i confronti necessari a processare un nodo (calcolati da `calcminchildren` e `calcminparents`), cioè in cui a è $|V_T^*|$ e $b \in O(|V_T^*|)$;

la complessità in tempo risulta quella nella figura 6: $O(|T|) + P(|V_T^*|)$. P è una funzione espressa dall'equazione ricorsiva

$$P(z) = \begin{cases} \Theta(1) & \text{se } z = 1 \\ f(z) + P(z-1) & \text{altrimenti,} \end{cases}$$

in cui f è una funzione definita per tutti gli $m \in \{2, \ldots, |V_T^*|\}$ ed è lineare rispetto al numero di confronti necessari per togliere dalle liste l'$m - |V_T^*| + 1$-esimo nodo scelto come più conveniente: i possibili archi vengono cercati in due porzioni di ordinamenti contenenti m elementi, quindi $f(m) \in O(m)$.

Procedura	Esecuzioni	Complessità totale: tempo
`top_char`: ordinamento topologico e impilamento	$\lvert S\rvert$	$O(\lvert S\rvert \cdot (\lvert V_T^*\rvert + j_{\max})) = O(\lvert T\rvert)$
`pileup`: impilamento	1	$\Theta(\lvert V_T^*\rvert) = O(\lvert T\rvert)$
`findpointers`: indicizzazione liste	1	$\Theta(\lvert S\rvert \cdot \lvert V_T^*\rvert) = O(\lvert T\rvert)$
`calcminchildren`: calcolo liste migliori	$\lvert S\rvert + 1$	$\Theta((\lvert S\rvert + 1) \cdot \lvert V_T^*\rvert) = O(\lvert T\rvert)$
`calcminparents`	$\lvert S\rvert + 1$	$O(\lvert T\rvert)$
`queueup`: ordinamento secondo convenienza	1	$\Theta(\lvert V_T^*\rvert + \lvert V_T^*\rvert) = O(\lvert T\rvert)$
`calcchildren` e `calcparents`: calcolo degli archi	$\lvert V_T^*\rvert$	$P(\lvert V_T^*\rvert)$
`topsremove`: rimozione dalle liste	$\lvert V_T^*\rvert$	$\Theta((\lvert S\rvert + 1) \cdot \lvert V_T^*\rvert) = O(\lvert T\rvert)$
Totale		$O(\lvert T\rvert) + P(\lvert V_T^*\rvert)$

Figura 6: Complessità in tempo del calcolo degli archi.

Caso peggiore Nel caso peggiore, $f(m) \in \Theta(m)$ per ogni $m \in \{2, \ldots, |V_T^*|\}$, quindi l'equazione ricorsiva viene approssimata da

$$P_p(z) = \begin{cases} c_1 & \text{se } z = 1 \\ c_z z + P(z-1) & \text{altrimenti,} \end{cases}$$

con c_i per $i \in \{1, \ldots, |V_T^*|\}$ costanti. Allora

$$P_p(z) = c_z z + c_{z-1}(z-1) + \cdots + c_1 = c \cdot \left(\frac{z(z+1)}{2} \right) \in \Theta(z^2);$$

quindi, dato n il numero di parole in T, ricordando il teorema 4.2 e notando che $|V_T^*| \leq |V_T| \leq n$, la complessità in tempo del calcolo degli archi nel caso peggiore è

$$O(|T|) + P_p(|V_T^*|) = O(|T|) + \Theta(|V_T^*|^2) = O(|T|^2).$$

Caso ottimo Il caso migliore invece si verifica quando $f(m) \in \Theta(1)$ per ogni m. L'equazione diventa

$$P_o(z) = \begin{cases} d_0 & \text{se } z = 1 \\ d_z + P(z-1) & \text{altrimenti,} \end{cases}$$

con d_i per $i \in \{0, \ldots, |V_T^*|\}$ costanti. Allora

$$P_o(z) = d_z + d_{z-1} + \cdots + d_1 + d_0 = d \cdot z \in \Theta(z);$$

la complessità totale in tempo diventa

$$O(|T|) + P_o(|V_T^*|) = O(|T|) + \Theta(|V_T^*|) = O(|T|).$$

4.8 Calcolo della lunghezza del percorso massimo

`alg_findllp` scandisce i nodi di G_T^* dalla fine dell'ordinamento topologico secondo cui sono già ordinati e per ogni nodo calcola la lunghezza massima dei percorsi che partono da esso.

Intuitivamente ogni percorso di G_T corrisponde ad un percorso di G_T^* caratterizzato dagli stessi vettori dei caratteri, e ogni percorso di G_T^* corrisponde ad almeno un percorso di G_T; infatti sono stati raggruppati tra di loro nodi equivalenti (con gli stessi figli e gli stessi genitori). Quindi la massima lunghezza dei percorsi in G_T^* sarà anche quella di G_T.

4.8.1 Implementazione

src/algs/algs.c

```
0  void alg_findllp(struct graph *G_T)
1  {
2          struct dl_list_n *s; /* elemento della lista di V_T^* corrente */
3          struct node *x; /* nodo corrente */
4          struct dl_list_n *j; /*elemento della lista di adiacenza corrente*/
5          struct node *y; /* figlio di x corrente */
6          int c; /* lunghezza massima corrente */
7
8          s = dlln_last(G_T->V);
9          while (!dlln_isempty(s)) {
10                 x = dlln_get(s);
11
12                 node_setllp(x,0);
13
14                 j = node_adj(x);
15                 while (!dlln_isempty(j)) {
16                         y = dlln_get(j);
17
18                         c = node_getllp(y);
19                         if (c >= node_getllp(x))
20                                 node_setllp( x, c+1 );
21
22                         j = dlln_next(j);
23                 }
24
25                 s = dlln_prev(s);
26         }
27 }
```

4.8.2 Correttezza

Lemma 4.5. Dato $G = (V, E)$ orientato e aciclico, sia $g : V \to \mathbb{N}$ tale che $g(x)$ è la massima lunghezza dei percorsi uscenti da $x \in V$ (sia $g(x) = 0$ se x è una foglia). Allora vale che

$$g(x) = \begin{cases} 0 & \text{se } \mathrm{Adj}[x] = \emptyset \\ \max\limits_{y \in \mathrm{Adj}[x]} \Big(g(y)\Big) + 1 & \text{altrimenti} \end{cases}$$

Dimostrazione. Il problema è ben posto, poiché il grafo è aciclico: la massima lunghezza dei percorsi uscenti da qualunque nodo è un intero minore di $|V|$. La dimostrazione procede per induzione sulla massima lunghezza l dei percorsi

uscenti dal nodo generico x; l'invariante è la validità della tesi per tutti i nodi con massima lunghezza l.

casi base: se $l = 0$, cioè non ci sono percorsi uscenti, x è una foglia, $\text{Adj}[x] = \emptyset$ e quindi per costruzione $g(x) = 0$;
se $l = 1$, tutti i percorsi uscenti hanno come secondo vertice una foglia, cioè $\forall y \in \text{Adj}[x]\ \ \text{Adj}[y] = \emptyset$ e $g(y) = 0$, quindi $g(x) = 1$.

passo induttivo: se $l > 1$, sia vera l'invariante per tutti i $k < l$; tutte le lunghezze dei percorsi uscenti dai figli di x sono minori di l, quindi su di essi si puó applicare l'ipotesi induttiva; esistono uno o più figli di x con percorso massimo uscente lungo $l-1$ e per ipotesi induttiva la funzione g per essi assume il valore $l - 1$; allora, preso $g(y)$ come massimo, con $y \in \text{Adj[x]}$, vale che

$$g(x) = g(y) + 1 = l - 1 + 1 = l.$$

Quindi l'invariante è vera per qualunque lunghezza massima dei percorsi uscenti, cioè per qualunque nodo di G. □

Teorema 4.6. Al termine di `alg_findllp(G_T)`, per ogni nodo `x` di `G_T->V`, `x->llp` è il valore della lunghezza massima dei percorsi uscenti da `x`.

Dimostrazione. L'algoritmo ad ogni nodo assegna il valore indicato dalla funzione g del lemma 4.5; in particolare, poiché i nodi vengono scanditi in ordine topologico inverso, non c'è bisogno di calcolarla ricorsivamente. Quindi, per lo stesso lemma, `x->llp` è la lunghezza massima dei percorsi uscenti da `x`. □

4.8.3 Complessità

Per ogni nodo di G_T^* viene scandita la sua lista di adiacenza: `alg_findllp` ha complessità in tempo $\Theta(|V_T^*| + |E_T^*|)$. La complessità in spazio è $\Theta(|V_T^*|)$, poiché a ogni nodo viene assegnato un intero.

4.9 Stampa dei risultati

`print_mllp` stampa il numero trovato da `graph_findllp` e ha complessità costante sia in tempo che in spazio. `print_DOT` invece ricostruisce al volo G_T da G_T^* e dagli anagrammi scomparsi salvati (una dimostrazione di correttezza si baserebbe sul teorema 2.14); ha complessità in tempo $\Theta(|V_T| + |E_T|)$ e in spazio $\Theta(|V_T^*|)$, poiché assegna ad ogni vertice di G_T^* un indice per il grafo in formato DOT (gli indici di G_T degli anagrammi vengono dedotti).

4.10 Complessità complessiva

Riprendendo le procedure principali, la complessità totale dell'algoritmo risolutivo, come mostrato nella figura 7, è $O(|T|^2)$ sia in tempo che in spazio.

Procedura	Complessità: tempo	spazio
`text_retrieve`	$\Theta(\|T\|)$	$O(\|T\|)$
`alg_removeduplicates`	$\Theta(n) = O(\|T\|)$	$O(\|T\|)$
`graph_retrievenodes`	$\Theta(\|V_T\|) = O(\|T\|)$	$O(\|T\|)$
`graph_joinanagrams`	$\Theta(\|V_T\|) = O(\|T\|)$	$O(\|T\|)$
`graph_topologicalsort`	$O(\|V_T^*\| + \|T\|) = O(\|T\|)$	$O(\|T\|)$
`graph_findedges`	$O(\|V_T^*\|^2) = O(\|T\|^2)$	$O(\|T\|^2)$
`graph_findllp`	$\Theta(\|V_T^*\| + \|E_T^*\|) = O(\|T\|^2)$	$\Theta(\|V_T^*\|) = O(\|T\|)$
`print_mllp`	$\Theta(1)$	$\Theta(1)$
`print_DOT`	$\Theta(\|V_T\| + \|E_T\|) = O(\|T\|^2)$	$\Theta(\|V_T^*\|) = O(\|T\|)$
Totale	$O(\|T\|^2)$	$O(\|T\|^2)$

Figura 7: Complessità totale; n è il numero di parole in T.

5 Calcolo dei tempi

Per calcolare i tempi sono stati implementati gli algoritmi descritti dalla dispensa (la versione del 19 dicembre 2016) con le seguenti modalità o modifiche.

Precisione

- per la misurazione di un qualunque intervallo di tempo con l'orologio di macchina si è scelto un errore relativo rispetto alla granularità del sistema minore del 2% (l'opzione si trova in `src/tcalc/talgs/options.h`);
- nella stima del tempo medio di esecuzione, riguardo al coefficiente confidenza $1-\alpha$, è stato scelto α pari a $0,05$, corrispondente a $z(1-\frac{\alpha}{2}) = 1,96$ (in `src/tcalc/tcalc.c`);
- per la scelta dell'ampiezza massima tollerata per l'intervallo di confidenza (2Δ), l'Algoritmo 9 è stato modificato perché Δ sia al massimo il 5% del valore stimato (in `src/tcalc/tcalc.c`);

Ripetizioni in base alla granularità L'Algoritmo 5 delle dispense, che calcola il numero di ripetizioni necessarie perché l'errore effettuato dall'orologio di macchina nella misurazione sia piccola relativamente al tempo calcolato (è importante su input piccoli), è stato modificato in modo che il minor numero di ripetizioni possa essere 1.

La tara è trascurabile Il calcolo della tara per il suo scorporamento nel calcolo del tempo medio di esecuzione dell'algoritmo risolutivo su uno stesso input è stato saltato: l'algoritmo risolutivo legge il testo da *standard input* quindi non è in-place. Per la misurazione, allo *standard input* viene sostituito il file di input, preparato in precedenza, aperto in sola lettura; per ripreparare una nuova esecuzione sugli stessi dati è sufficiente una chiamata trascurabile[8] a `lseek` per impostare l'*offset* del file a 0.

[8]La complessità in realtà dipende dall'implementazione del sistema operativo e dal *filesystem* in cui risiede il file; ad esempio, per Linux e ext4 è $\Theta(1)$.

Preparazione del calcolatore I test sono stati eseguiti su un PC con 8GB di RAM, con il processore i5-7400 bloccato a 1GHz e utilizzando il sistema operativo Arch Linux a 64-bit: è stata scelta una frequenza così bassa per evitare che la gestione della memoria da parte del sistema operativo influisse sui tempi calcolati, causando irregolarità dovute a fattori esterni.

Interpolazione dei risultati Per mostrare l'apparente andamento dei risultati sperimentali viene mostrata la regressione lineare di `pgfplots` e la regressione creata dalla funzione `fit` di `gnuplot` per le funzioni non lineari indicate nelle legende.

5.1 Caso 1: ogni carattere è casuale

Può essere interessante misurare i tempi dell'algoritmo su una distribuzione in cui, dato l'alfabeto Σ, ogni suo carattere ha la stessa probabilità di occorrere per ogni carattere del testo generato. La grandezza dell'input sarà misurata in numero caratteri.

5.1.1 Costruzione della distribuzione e del campione

- è stato scelto a priori un alfabeto finito contenente lo spazio *blank* e altri 21 caratteri (l'alfabeto italiano);
- con il PRNG viene estratto un carattere casuale dell'alfabeto e scritto nel file in preparazione;
- viene ripetuto il passo precedente fino ad ottenimento di un testo della lunghezza desiderata.

5.1.2 Risultati

Grafico di $|V_T|$

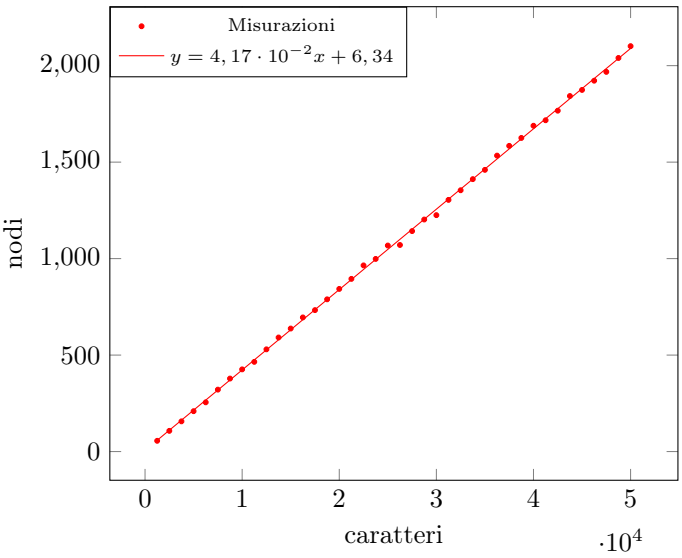


Grafico di $|E_T|$

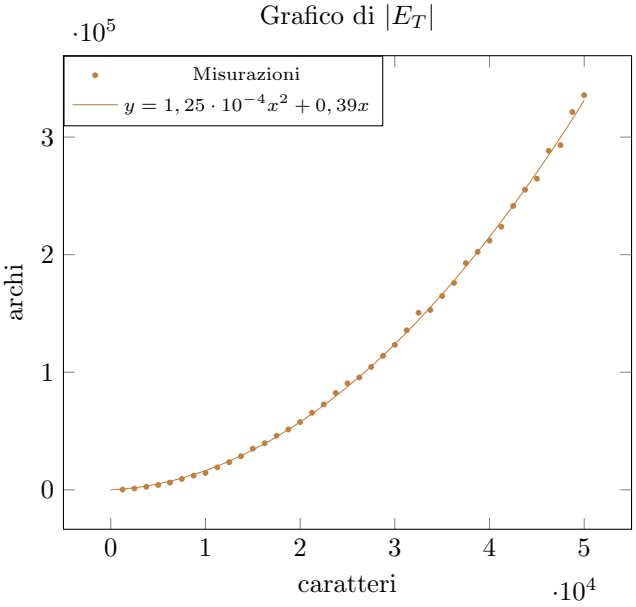
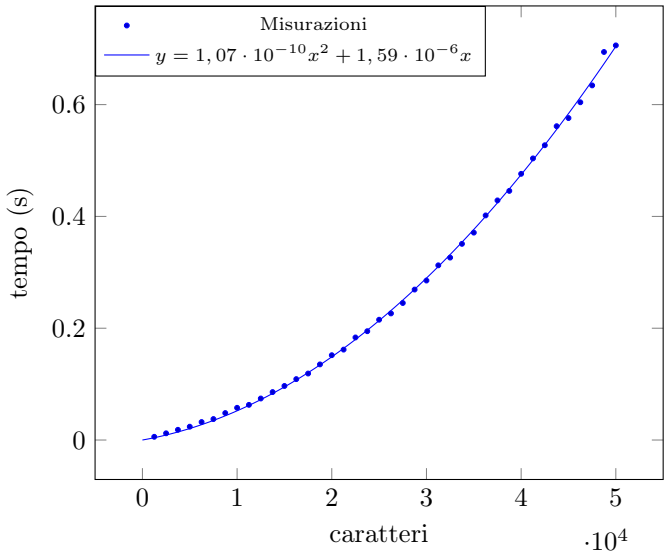


Grafico dei tempi



Presi in considerazione i grafici di $|V_T|$ e di $|E_T|$, l'algoritmo può essere visto come **mediamente ottimale** per questa distribuzione: qualunque algoritmo che debba trovare e stampare G_T nella sua interezza deve avere complessità mediamente quadratica rispetto al numero di caratteri, poiché E_T sembra crescere quadraticamente.

5.2 Caso 2: un testo senza archi

Esiste un caso medio per cui l'algoritmo risolutivo non è mediamente ottimale? La risposta è sì, e si può mostrare costruendo una distribuzione in cui G_T non ha archi. La grandezza dell'input sarà misurata sul numero di parole del testo.

5.2.1 Costruzione della distribuzione e del campione

La costruzione procede secondo i seguenti passaggi:

- il programma contenuto in `other/case_two_gen`[9] si occupa di generare la base della distribuzione:
  - viene scelto un alfabeto A non contenente il carattere ƀ (*blank*) e non contenente il carattere _ (*underscore*);
  - vengono generate tutte le parole lunghe $|A|$ contenenti $\lfloor |A|/2 \rfloor$ *underscore* e con i restanti caratteri corrispondenti ai caratteri di A dello stesso indice (ad esempio, per $A = \{a, b, c\}$, le parole sono $ab\_$, a_c e $\_bc$);
  - a queste parole vengono concatenati un numero variabile (ma piccolo) di asterischi ($*$), in modo che esse siano distribuite abbastanza uniformemente in lunghezze diverse;

[9] Un solo file di codice compilabile direttamente: `$clang case_two_gen.c -o c2gen`

- alle parole vengono tolti i caratteri *underscore* (poiché sono inutili) e vengono usate per generare `corpus2.txt`, un testo di righe del formato `parola`+`TAB`+`1`, dove `1` rappresenta la frequenza;
- dato il numero di parole del testo da generare, tramite estrazione a lotteria con reinserimento[10] con il PRNG viene estratta una parola alla volta e scritta nel testo in preparazione, fino al raggiungimento delle parole volute.

Ad esempio, le parole generate con $A = \{a, b, c, d\}$, concatenando da 0 a 2 asterischi e togliendo poi gli *underscore* sono:

- ab
- bc
- $ac*$
- $bd*$
- $ad**$
- $cd**$

Con la possibilità di scegliere A, i testi generabili sono interessanti per l'algoritmo risolutivo? Il numero di parole diverse è $\binom{|A|}{\lfloor |A|/2 \rfloor}$: nella costruzione vengono scelte tutte le parole contenenti $\lfloor |A|/2 \rfloor$ *underscore* e i restanti caratteri diversi.[11]

Teorema 5.1. Dato $x \in \mathbb{N}$, $\binom{x}{\lfloor x/2 \rfloor} \in \Omega(2^x)$.

Dimostrazione. Se x è pari, il binomiale di x su $\lfloor x/2 \rfloor$ è

$$\frac{x!}{\frac{x}{2}! \cdot \frac{x}{2}!} = \frac{x \cdot (x-1) \cdot \ \dots \ \cdot (\frac{x}{2}+2) \cdot (\frac{x}{2}+1)}{\frac{x}{2} \cdot (\frac{x}{2}-1) \cdot \ \dots \ \cdot 2 \cdot 1};$$

i prodotti si possono scomporre in

$$\frac{x}{\frac{x}{2}} \cdot \frac{x-1}{\frac{x}{2}-1} \cdot \dots \cdot \frac{\frac{x}{2}+2}{\frac{x}{2}-\frac{x}{2}+2} \cdot \frac{\frac{x}{2}+1}{\frac{x}{2}-\frac{x}{2}+1} = \prod_{i=0}^{\frac{x}{2}-1} \frac{x-i}{\frac{x}{2}-i} \geq \prod_{i=0}^{\frac{x}{2}-1} \frac{x-i}{\frac{x}{2}-i+\frac{i}{2}} = 2^{\frac{x}{2}},$$

quindi $\binom{x}{\lfloor x/2 \rfloor} \geq 2^{\frac{x}{2}}$.
Nel caso in cui x sia dispari, si può trovare la seguente relazione con il binomiale successivo:

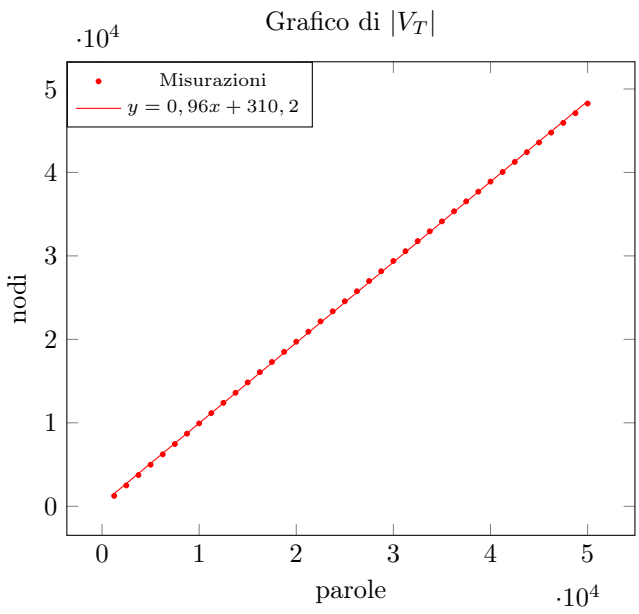
$$\binom{x+1}{\frac{x+1}{2}} = \frac{(x+1)!}{\frac{x+1}{2}!\frac{x+1}{2}!} = \frac{x+1}{\frac{x+1}{2}} \cdot \frac{x!}{(\frac{x+1}{2}-1)! \cdot \frac{x+1}{2}!} = 2 \cdot \binom{x}{\lfloor x/2 \rfloor}.$$

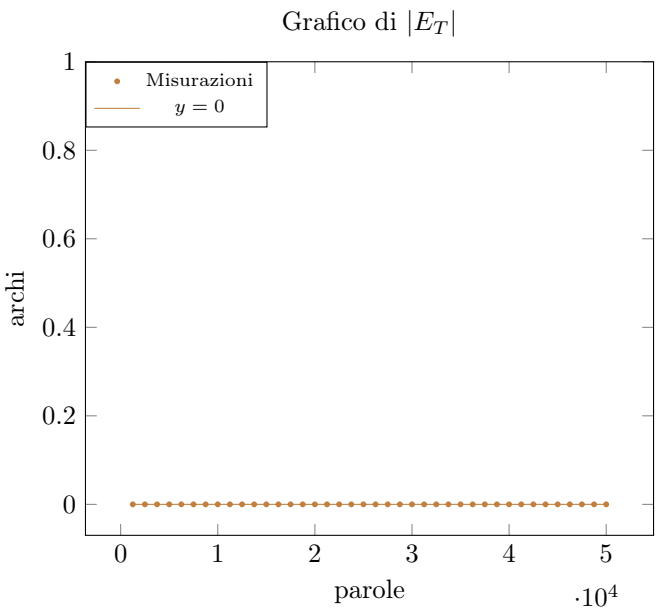
Perciò $\binom{x}{\lfloor x/2 \rfloor} \geq 2^{\frac{x}{2}-1}$.
Quindi, nel caso generale, $\binom{x}{\lfloor x/2 \rfloor} \in \Omega(2^x)$. □

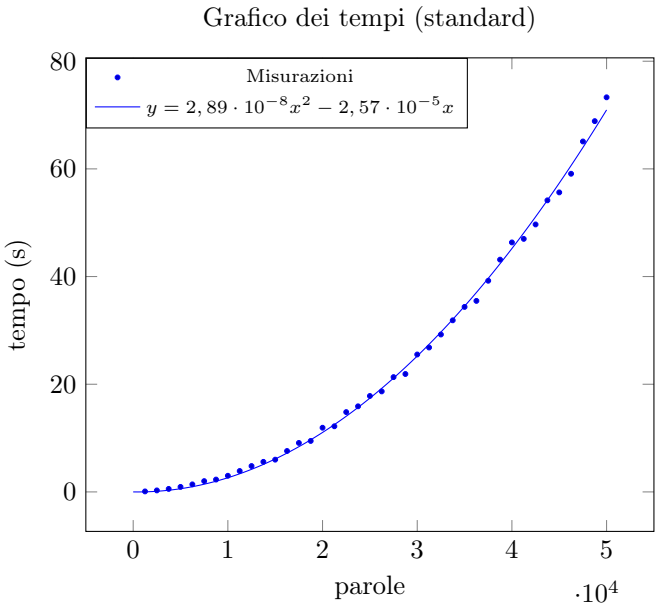
Cioè si può sempre scegliere un alfabeto A abbastanza grande per non preoccuparsi di finire le parole da estrarre: è stato scelto un alfabeto di 22 caratteri (l'alfabeto inglese dalla a alla w) e da 0 a 3 asterischi da concatenare.

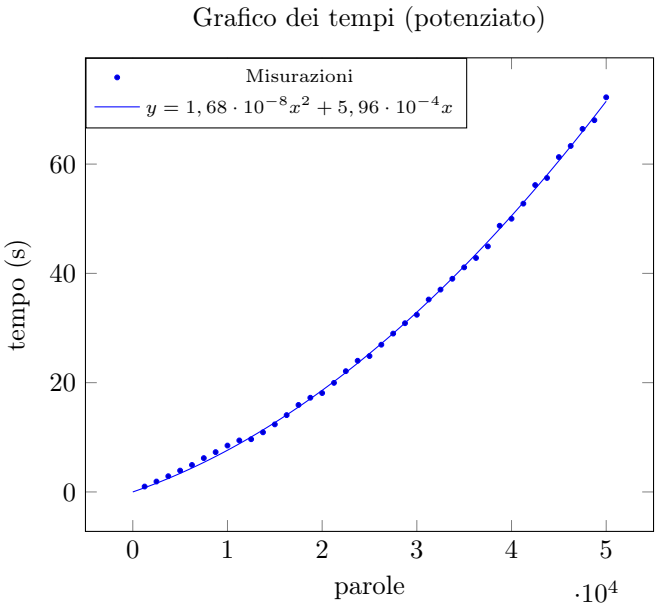
[10] Ciò rende possibile la presenza di doppioni, però come sarà verificato sperimentalmente $|V_T|$ cresce linearmente rispetto al numero di parole estratte.
[11] Dovrebbe essere il massimo numero di vettori dei caratteri non in relazione tra di loro per il teorema di Sperner (si veda [3]), considerandoli un insieme di sottoinsiemi di A tali che nessuno di questi ultimi è contenuto in un altro.

5.2.2 Risultati









Come si può vedere, $|V_T| + |E_T|$ cresce linearmente rispetto al numero di parole, mentre **i grafici dei tempi sembrano avere un andamento quadratico**. Neanche l'utilizzo dell'algoritmo potenziato per il calcolo degli archi migliora la complessità asintotica, mentre peggiorano i tempi nell'intervallo misurato: però il coefficiente del termine di secondo grado della funzione interpolata diminuisce.

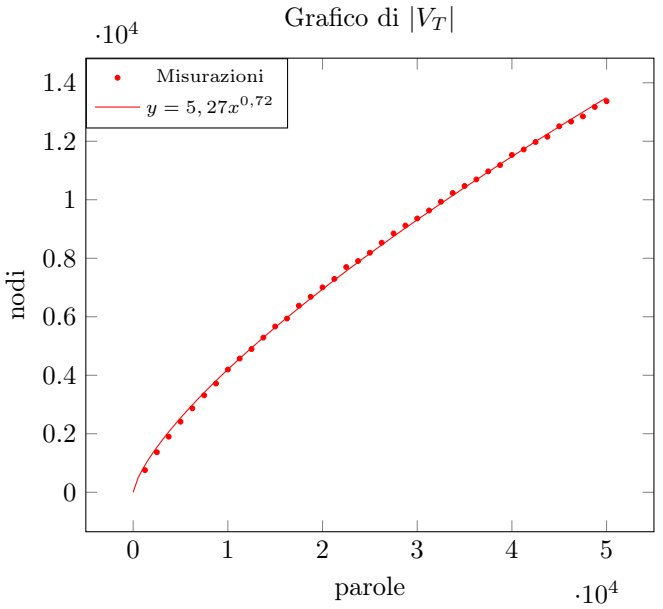
5.3 Caso 3: un testo di parole in italiano

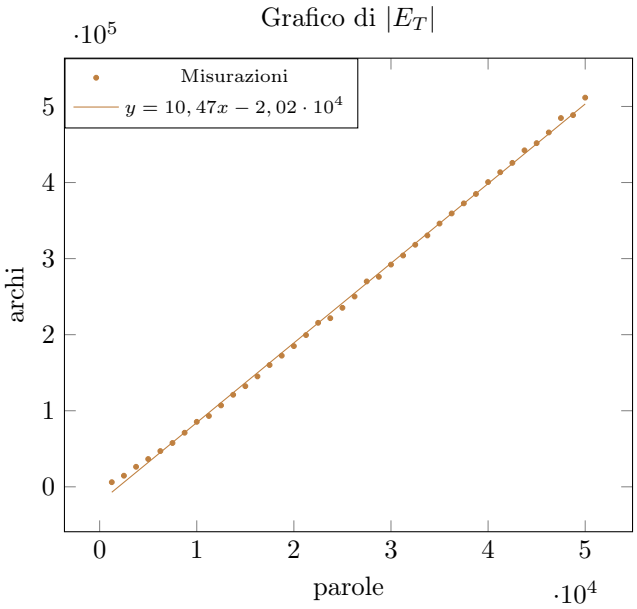
Come si comporta l'algoritmo su un testo di un linguaggio naturale, come l'italiano scritto? La grandezza dell'input sarà misurata sul numero di parole del

testo.

5.3.1 Costruzione della distribuzione e del campione

- è stato scaricato da `http://www.istc.cnr.it/grouppage/colfis` il formario del CoLFIS[12];
- con qualche comando da terminale e con l'aiuto di un editor di testo per ricodificare il testo in UTF-8 è stato generato `corpus3.txt`, un testo che per ogni forma del formario ha la riga `forma`+`TAB`+`frequenza`;
- dato il numero di parole del testo da generare, tramite estrazione a lotteria con reinserimento, viene estratta una parola alla volta e scritta nel testo in preparazione (separata dalle altre parole da un singolo *blank*), fino al raggiungimento del numero di parole desiderato.





[12] Un *corpus* della lingua italiana che rispecchia quotidiani, periodici e libri riscontrati nelle letture degli italiani nel periodo 1992-1994 secondo l'ISTAT (si vedano [4][5]).

Grafico dei tempi (standard)

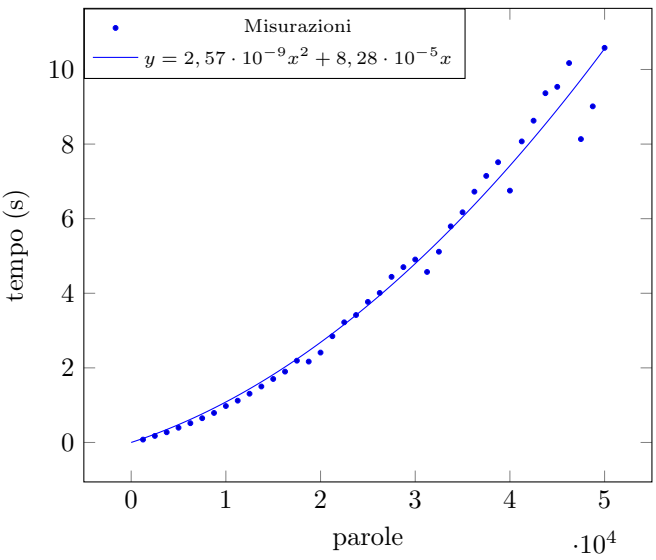
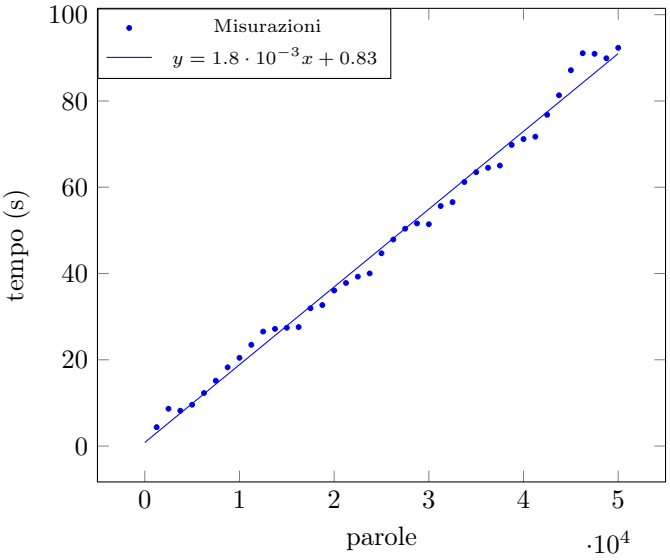


Grafico dei tempi (potenziato)



Anche in questo caso $|V_T| + |E_T|$ sembra crescere al massimo linearmente, ma l'andamento dei tempi è più irregolare. I tempi della versione standard sembrano quadratici, **la versione potenziata** peggiora localmente ma **sembra crescere meno che quadraticamente rispetto all'input**, o almeno ha il coefficiente di secondo grado della funzione interpolante molto più piccolo che nella versione standard.

6 Conclusioni

I risultati ricavati dalla misurazione sperimentale dei tempi nei tre casi medi non smentiscono il limite asintotico superiore ottenuto nell'analisi della complessità: l'algoritmo è in quasi tutti i casi **quadratico** rispetto al testo di input. Inoltre, teoria e pratica assieme identificano la generazione di E_T^* come la procedura più importante computazionalmente.

Tuttavia lo studio del problema e la progettazione di un buon algoritmo risolutivo sembrano lontani dall'essere i migliori possibili:

- l'approssimazione dell'algoritmo goloso per il calcolo degli archi (sezione 4.7) è una buona approssimazione?
- generalizzando gli ordinamenti dei teoremi 2.20 e 2.19 a più di due caratteri fissati, utilizzando uno schema con tutte le coppie di ordinamenti possibili per ogni sottoinsieme dell'alfabeto $S = \Sigma \setminus \{\text{b}\}$ (sono circa $2^{|S|+1}$, che è costante), è possibile migliorare la complessità per il caso 2? e in un caso generico?
- sarebbe stato possibile sfruttare conoscenze approfondite di teoria degli ordini, dei grafi e dei reticoli?

Riferimenti bibliografici

[1] Cormen, Thomas H., Leierson, Charles E., Rivest, Ronald L., Stein, Clifford (2010), *Introduzione agli algoritmi e alle strutture dati*, Terza edizione, McGraw-Hill, Milano

[2] Wikipedia (2017), *Relazione d'ordine*, `https://it.wikipedia.org/wiki/Relazione_d'ordine`

[3] Wikipedia (2017), *Sperner's theorem*, `https://en.wikipedia.org/wiki/Sperner%27s_theorem`

[4] Baroni, Marco (2010), *Enciclopedia dell'Italiano:* corpora *di italiano*, Treccani, `http://www.treccani.it/enciclopedia/corpora-di-italiano_(Enciclopedia-dell'Italiano)/`

[5] Bertinetto, Pier Marco, Burani, Cristina, Laudanna, Alessandro, Marconi, Lucia, Ratti, Daniela, Rolando, Claudia, Thornton, Anna Maria (2005), *Corpus e lessico di frequenza dell'italiano scritto (CoLFIS)*, `http://linguistica.sns.it/CoLFIS/Home.htm`