05325A Manuel Marrali

PROGETTO "PAROLE E CATENE DI PAROLE", LABORATORIO DI ALGORITMI E STRUTTURE DATI

Il programma realizzato si pone l'obiettivo di gestire un dizionario contenente sia parole (sequenze di caratteri minuscoli appartenenti all'alfabeto inglese), sia schemi (sequenze di caratteri appartenenti all'alfabeto inglese in cui è presente almeno una lettera maiuscola). Il dizionario permette di eseguire diverse operazioni di **gestione**, **ricerca** e **confronto** tra parole e schemi, secondo quanto richiesto dalla traccia progettuale.

La modellazione del dizionario avviene mediante una struttura dati denominata **Dizionario**, composta da due **mappe** distinte, definita come segue:

```
type Dizionario struct {
   parole map[string]struct{}
   schemi map[string]struct{}
}
```

Sia il campo **parole** che il campo **schemi** è una mappa che associa stringhe (parole o schemi) a valori di tipo struct{}.

La scelta di utilizzare una mappa con valore vuoto *struct{}* è motivata dall'esigenza di rappresentare un insieme di stringhe in modo efficiente: il tipo *struct{}* in Go occupa zero byte, pertanto questa soluzione risulta più leggera in termini di memoria rispetto all'uso di una mappa con valori booleani (*map[string]bool*). In quest'ultimo caso, infatti, ogni elemento richiederebbe spazio aggiuntivo per memorizzare il valore booleano, mentre con *struct{}* la presenza della chiave nella mappa è sufficiente per indicare l'appartenenza all'insieme, inoltre tramite l'istruzione *_, exists := m[key]* è possibile verificarlo.

In virtù di ciò, l'implementazione di "*v := make(set[string])*" risulterebbe più chiara e potrebbe fungere da zucchero sintattico rispetto a *map[string]struct{}*

Non è stato necessario utilizzare puntatori a mappe (**map[string]struct{}*) in quanto, nel linguaggio Go, le mappe sono già di per sé tipi di riferimento. Passandole per valore si trasferisce infatti solo il riferimento alla struttura interna, evitando copie costose e mantenendo la semplicità nell'uso, senza la necessità di puntatori specifici.

L'alias di tipo **dizionario**

```
type dizionario *Dizionario
```

è stato introdotto per agevolare la gestione del dizionario (Dizionario) nelle funzioni e per mantenere coerente la segnatura proposta dal progetto, rendendo esplicito il passaggio di un riferimento alla struttura.

Questa scelta migliora la leggibilità del codice e sottolinea che le operazioni vengono effettuate direttamente sull'istanza originale, evitando copie inutili della struttura dati.

Questa combinazione di scelte garantisce un accesso rapido ed efficiente alle parole e agli schemi, con operazioni di inserimento, cancellazione e verifica che hanno complessità media di O(1).

Le operazioni principali sul dizionario sono implementate tramite funzioni dedicate, gestite da una funzione centrale di controllo chiamata **esegui()**. Questa funzione riceve in input il dizionario e la stringa di comando da eseguire, la scompone nei singoli elementi e richiama l'operazione corrispondente, seguendo una struttura di controllo *switch*.

Inizializzazione e creazione del dizionario

La funzione **crea()** si occupa di inizializzare un dizionario esistente, azzerandone le mappe interne parole e schemi. In pratica, assegna a ciascuna mappa una nuova mappa vuota tramite la funzione **make**, garantendo così un dizionario completamente vuoto pronto all'uso.

Per la creazione di un dizionario invece si utilizza la funzione **newDizionario()** che costruisce e restituisce un puntatore (dizionario) a una nuova istanza di Dizionario con mappe vuote inizializzate grazie alla funzione precedente. Questa funzione è utile per creare un dizionario da zero e passarlo alle altre funzioni tramite il tipo alias **dizionario**.

Gestione dei comandi

La funzione **esegui()** interpreta i comandi in ingresso seguendo una sintassi a token, scompone la stringa di input in parole tramite *strings.Fields* e gestisce i vari casi indicati nella "Tabella 1: Specifiche del programma" della consegna.

Per ogni comando, **esegui()** controlla la correttezza del formato (numero di argomenti) e in caso di errore stampa un messaggio esplicito. I comandi non riconosciuti sono intercettati e segnalati all'utente.

Scelte implementative

L'approccio adottato con la funzione **esegui()** centralizza la gestione di tutte le operazioni, favorendo modularità e facilità di estensione. Le funzioni di supporto (**crea()**, **inserisci()**, **elimina()**) sono progettate per operare direttamente sulle mappe del dizionario, modificandole in modo efficiente.

L'uso di mappe per parole e schemi consente di effettuare operazioni di inserimento, cancellazione e verifica con complessità O(1), garantendo buone prestazioni anche per dizionari di dimensioni elevate.

Prima di affrontare l'analisi degli algoritmi utilizzati riguardanti le funzioni **distanza()** e **catena()**, si pone l'attenzione su uno studio di ricerca intitolato: "**Average word length dynamics as indicator of cultural changes in society**", nel quale a seguito di un'analisi approfondita, si evince che la lunghezza media delle parole varia da 4.3 a 5.1 caratteri.

Link dell'articolo:
https://www.researchgate.net/publication/230764201_Average_word_length_dynamics_as_indicator_of_cultural_changes_in_society

Tenendo conto delle informazioni consultate tramite lo studio di ricerca e valutando i set di dati che il programma dovrà gestire, si analizzano di seguito di algoritmi implementati.

In primo luogo, la funzione **distanza()**, che utilizzando l'algoritmo della distanza di Levenshtein, permette il calcolo della distanza di editing

L'algoritmo, di fatto, calcola la distanza di editing tra due stringhe, cioè il numero minimo di operazioni elementari (inserimento, cancellazione, sostituzione) necessario per trasformare una stringa nell'altra.

La versione implementata usa una programmazione dinamica ottimizzata in memoria, mantenendo solo due vettori (**prev** e **curr**) per la riga precedente e quella corrente della matrice di calcolo.

Questo riduce l'uso di memoria da O(m*n) a O(n), dove m e n sono le lunghezze delle due stringhe.

Complessità e prestazioni

- **Tempo**: O(m*n), questo perché l'algoritmo deve confrontare tutte le combinazioni di prefissi delle due stringhe.
- **Spazio**: O(n), grazie all'ottimizzazione dei due array.

Questa complessità è accettabile per le parole del dizionario che, come detto in precedenza, hanno lunghezza media compresa tra 4.3 e 5.1 caratteri. L'algoritmo è quindi efficiente in termini di tempo e spazio per l'uso previsto, è un metodo ben noto, stabile e facile da implementare correttamente e la versione ottimizzata minimizza l'uso di memoria.

Gli svantaggi riguardano la velocità per stringhe molto lunghe (ma non è questo il caso) e il calcolo della distanza esatta senza alcuna euristica, quindi sempre con complessità piena, motivo per cui si è tenuto in considerazione l'algoritmo di A*. Quest'ultimo è un algoritmo di ricerca su grafi che utilizza una funzione euristica per guidare la ricerca verso la soluzione più rapidamente. Tuttavia:

La distanza di Levenshtein non è una ricerca su un grafo esplicito di stati con un obiettivo finale facilmente euristico. È un problema di confronto di stringhe, per cui la programmazione dinamica è più diretta e semplice.

A* richiederebbe una definizione di spazio degli stati e una funzione euristica ammissibile per ogni operazione, cosa complessa e meno efficiente in questo contesto.

Avendo quindi distanze tra parole di lunghezza limitata, la programmazione dinamica è sufficientemente efficiente senza complicare la soluzione.

Per la funzione **catena()**, che ricerca la catena, si utilizza l'algoritmo **BFS** (Breadth-first search).

La ricerca di una catena tra due parole consiste nel trovare una sequenza di parole consecutive con distanza di editing 1. Questo è un problema di ricerca del percorso più breve in un grafo, dove ogni parola è un nodo e gli archi collegano parole con peso 1.

La funzione **catena()** usa la **BFS** per esplorare il grafo implicito delle parole. BFS garantisce che il primo percorso trovato sia anche il percorso minimo per numero di passaggi, cioè la catena più corta possibile.

Vengono mantenute: una coda per la gestione FIFO dei nodi da esplorare, una mappa di predecessori per ricostruire il percorso una volta raggiunta la parola finale ed una mappa di parole visitate per evitare cicli e visite ridondanti.

Complessità e prestazioni

Nel caso peggiore, BFS visita tutte le parole del dizionario.

Per ogni parola esplorata, si confrontano tutte le parole nel dizionario per verificarne la similitudine quindi, la complessità è circa $O(P^2 * L)$, dove P è il numero di parole nel dizionario e L è la lunghezza media delle parole (usata da *isSimile()*).

Anche se oneroso, conoscendo la lunghezza media delle parole e avendo usualmente dizionari non estremamente grandi, è una complessità accettabile.

BFS garantisce di trovare la catena più corta in termini di numero di parole, è semplice da implementare e da capire ed è deterministico: se esiste una catena, la troverà. Lo svantaggio è sicuramente la complessità elevata nel caso di dizionari molto grandi e lo spazio proporzionale al numero di parole visitate.

Si esclude DFS poiché esplora i percorsi profondamente e potrebbe trovare catene non minime, inoltre potrebbe entrare in cicli o esplorare percorsi lunghi prima di trovare la destinazione.

In questo problema si richiede la catena più corta (distanza = 1), che BFS trova in modo naturale.

La scelta di algoritmi classici e consolidati (Levenshtein e BFS) consente un buon bilanciamento tra complessità implementativa e performance. Per dizionari di piccola/media dimensione e parole di lunghezza breve, gli algoritmi sono efficienti e rispondono bene ai requisiti.

In contesti più grandi, si potrebbe migliorare con strutture dati più sofisticate o con algoritmi di ricerca più avanzati, ma a costo di maggiore complessità di implementazione.

RASSEGNA DI ESEMPI

Caso tipico: catena semplice tra parole simili

input:

i casa

i case

c casa case

output:

(

casa

case

)

Verifica che il programma trovi correttamente una catena di lunghezza 2 tra parole che differiscono per un solo carattere.

Caso limite: Catena tra la stessa parola

input:

i casa

c casa casa

output:

(

casa

)

Verifica che il programma gestisca correttamente il caso in cui le parole di partenza e arrivo coincidono, producendo la catena più semplice possibile.

Caso patologico: Nessuna catena possibile

input:

i casa

i mela

c casa mela

output:

non esiste

Due parole con lunghezze diverse o distanze di editing > 1 e nessuna catena possibile. Testa la capacità del programma di riconoscere l'impossibilità di una catena.

Caso limite: Dizionario molto grande con molte parole simili

input:

Un dizionario con migliaia di parole di 5 lettere, tutte molto simili (es. parole che differiscono per un solo carattere).

Si cerca una catena tra due parole distanti di 10 modifiche.

Testa l'efficienza dell'algoritmo BFS nella ricerca di catene lunghe in un grafo molto connesso. Questo caso evidenzia la complessità temporale e la gestione della memoria durante la ricerca.

Caso limite: Parole lunghe ma poche parole nel dizionario

input:

Parole molto lunghe (es. 20-30 caratteri) ma pochi elementi nel dizionario.

Matte alla prova la funzione **distanzaLevenshtein()** in situazioni in cui la complessità O(m*n) può diventare significativa anche se il grafo di parole è piccolo. Importante per misurare il costo computazionale della distanza di editing su parole lunghe.

Caso limite: Inserimento e rimozione dinamica

input:

Inserimento di molte parole, alcune rimozioni, e ricerche successive di catene e compatibilità.

Testa la corretta gestione dinamica del dizionario e verifica che la struttura dati mantenga coerenza ed efficienza dopo modifiche ripetute.

Caso particolare: Schemi con lettere maiuscole e minuscole

input:

Schemi con lettere maiuscole e minuscole.

Verifica la gestione della compatibilità tra parole e schemi con lettere maiuscole/minuscole, assicurandosi che la funzione di compatibilità funzioni correttamente secondo la definizione della consegna.

Questi casi mettono in evidenza:

- La correttezza e robustezza del programma in condizioni normali e limite
- L'efficienza dell'algoritmo BFS per trovare cammini minimi e il perché di questa scelta rispetto ad altre strategie (DFS)
- L'effetto della complessità della distanza di Levenshtein sul tempo di calcolo, soprattutto per parole lunghe
- La gestione dinamica del dizionario in inserimenti ed eliminazioni
- La corretta interpretazione degli schemi rispetto alle parole

05325A Manuel Marrali