Relazione Progetto "Parole"
Laboratorio di Algoritmi e Strutture Dati

Nome: Antonino
Cognome: Ottinà
Matricola: 24754A

Contents

1 Introduzione e Organizzazione del Codice

Questa relazione descrive le scelte progettuali e implementative adottate per la realizzazione del progetto "Parole". L'obiettivo del progetto è la creazione di un programma in Go per la gestione di un dizionario di parole e schemi, con funzionalità che vanno dall'inserimento e cancellazione alla ricerca di catene di parole.

Il programma è stato sviluppato in un unico file sorgente, solution.go, per semplicità e per mantenere tutta la logica applicativa in un unico posto. Il codice è stato strutturato in modo da separare la definizione della struttura dati principale (dizionario) dalle funzioni che operano su di essa e dalla funzione main che gestisce il ciclo di vita del programma e l'interazione con l'utente.

La compilazione del programma avviene tramite il comando standard di Go:

```
go build solution.go
```

L'eseguibile risultante, solution, può quindi essere utilizzato per eseguire i comandi letti da standard input, come specificato nel documento del progetto.

2 Modellazione del Problema e Scelte Progettuali

La modellazione del problema si è concentrata sulla definizione di strutture dati efficienti per rappresentare il dizionario e per supportare le operazioni richieste.

2.1 Struttura Dati del Dizionario

Il dizionario è stato modellato utilizzando una struct in Go chiamata dizionario. Questa struct contiene al suo interno due mappe (hash map) per memorizzare le parole e gli schemi:

```
type dizionario struct {
    words   map[string]struct{}
    schemes map[string]struct{}
}
```

La scelta di utilizzare le mappe è motivata dalle loro prestazioni eccellenti ($O(1)$ in media) per le operazioni di inserimento, cancellazione e ricerca di un elemento. Questo è fondamentale, dato che molte delle funzionalità del programma si basano sulla verifica rapida della presenza di una parola o di uno schema nel dizionario. Le chiavi delle mappe sono le parole o gli schemi stessi, mentre il valore è una struct vuota, un'idioma comune in Go per implementare un insieme (set) tramite una mappa, minimizzando l'uso di memoria.

2.2 Algoritmi Principali

Sono stati implementati diversi algoritmi per le funzionalità più complesse.

2.2.1 Calcolo della Distanza di Editing

Per calcolare la distanza di editing tra due parole (operazione distanza), è stato implementato l'algoritmo di Damerau-Levenshtein. Questo algoritmo, una variante di quello di Levenshtein, calcola il numero minimo di operazioni di inserimento, cancellazione, sostituzione e trasposizione di caratteri adiacenti necessarie per trasformare una stringa in un'altra. L'implementazione utilizza la programmazione dinamica, costruendo una matrice dp dove dp[i][j] rappresenta la distanza tra i primi **i** caratteri della prima parola e i primi **j** della seconda. La complessità di questo algoritmo è $O(m \cdot n)$, dove m e n sono le lunghezze delle due stringhe.

2.2.2 Ricerca di Catene di Parole

La ricerca di una catena di parole di lunghezza minima (operazione catena) tra due parole x e y è stata modellata come la ricerca di un cammino minimo in un grafo non pesato. Il grafo ha per nodi tutte le parole presenti nel dizionario e un arco tra due nodi (parole) se la loro distanza di editing è 1 (cioè sono "simili").

Per trovare il cammino minimo, è stato utilizzato l'algoritmo di **Ricerca in Ampiezza (BFS)**. Partendo dalla parola iniziale x, il BFS esplora il grafo a livelli, garantendo di trovare il percorso più breve verso la parola di destinazione y. La coda del BFS memorizza i percorsi parziali. La complessità è $O(V + E)$, dove V è il numero di parole nel dizionario e E è il numero di relazioni "simili" tra di esse. Nel caso peggiore, questo si traduce in $O(N^2 \cdot L)$, dove N è il numero di parole e L è la complessità del calcolo della distanza.

3 Dettagli Implementativi

Il programma è stato implementato seguendo le specifiche fornite.

3.1 Funzioni Principali

Come richiesto, sono state definite le seguenti funzioni:

- newDizionario() dizionario: Questa funzione implementa l'operazione crea(), inizializzando e restituendo una nuova istanza della struct dizionario con le mappe pronte per l'uso.
- esegui(d dizionario, s string): Questa funzione riceve una stringa di comando da standard input e applica l'operazione corrispondente sul dizionario. Dopo aver fatto il parsing della riga, la funzione seleziona dinamicamente quale metodo invocare tramite un'istruzione switch. Le verifiche sulla validità del comando e sul numero di argomenti permettono di gestire errori in modo robusto, il costo computazionale è lineare nella lunghezza della stringa di input, più quello dell'operazione effettiva eseguita.
- carica(file): La funzione carica(filename) implementa l'operazione carica, leggendo da un file di testo il contenuto e inserendo automaticamente nel dizionario tutte le parole e gli schemi trovati. L'intero contenuto del file viene letto in memoria e suddiviso in token tramite strings.Fields, così da supportare spazi, tabulazioni e newline come separatori. Ogni token viene quindi passato a inserisci, che lo memorizza nella mappa corrispondente. Dal punto di vista computazionale, l'operazione ha complessità O(B), dove B è la dimensione in byte del file, e O(N) per l'inserimento dei N token letti. Il costo totale è quindi O(B + N).
- stampaParole() e stampaSchemi(): Queste funzioni stampano rispettivamente l'insieme delle parole e degli schemi presenti nel dizionario. Gli elementi vengono prima raccolti in slice ([]string), poi ordinati alfabeticamente tramite sort.Strings, e infine stampati uno per riga tra parentesi quadre. Questo garantisce un output deterministico. L'ordinamento ha complessità O(N log N), dove N è il numero di elementi da stampare. La stampa richiede O(N), per un costo complessivo O(N log N).
- ricerca(S): La funzione ricerca stampa tutte le parole compatibili con lo schema S, secondo le regole definite nel testo del progetto. Per ogni parola nel dizionario, viene verificata la compatibilità con S tramite una funzione che simula un'assegnazione coerente tra lettere maiuscole e minuscole. La complessità dell'operazione è O(N·L), dove N è il numero di parole nel dizionario e L è la lunghezza media delle parole.

Il main del programma gestisce un ciclo di lettura dallo standard input. Ogni linea letta viene passata alla funzione esegui. Il programma termina quando viene letto il comando 't'.

3.2 Gestione dell'Output

L'output è formattato come specificato nel documento del progetto. Per gli insiemi di parole o

schemi, gli elementi vengono prima raccolti in una slice, ordinati alfabeticamente usando la funzione sort.Strings di Go, e infine stampati uno per riga, racchiusi tra parentesi quadre. Questo garan- tisce un output deterministico e ordinato, facilitando il testing e la verifica della correttezza, anche se l'ordine non era un requisito esplicito. Le catene, invece, vengono stampate nell'ordine corretto racchiuse tra parentesi tonde.

4 Testing e Collaudo

Per garantire la correttezza del programma, è stato effettuato un collaudo approfondito, utilizzando sia i test forniti sia esempi aggiuntivi progettati per coprire casi limite e situazioni particolari.

4.1 Casi di Test Significativi

Di seguito una rassegna di alcuni casi di test considerati:

1. **Dizionario Vuoto**: Verificare che tutte le operazioni si comportino correttamente quando il dizionario è vuoto.
2. **Input Duplicati**: Testare l'inserimento multiplo della stessa parola o schema per assicurarsi che non vengano creati duplicati, come da specifiche.
3. **Catene**:
   - **Catena Inesistente**: Richiedere una catena tra due parole che appartengono a gruppi disgiunti o che non sono nel dizionario. L'output atteso è "non esiste".
4. **Compatibilità tra Schemi e Parole**:
   - Uno schema del tipo ABc e una parola aac. L'assegnazione $\sigma(A) = a, \sigma(B) = a$ è valida.
   - Uno schema ABc e una parola adc. L'assegnazione $\sigma(A) = a, \sigma(B) = d$ è valida.
   - Uno schema AAc e una parola bbc. La parola non è compatibile perché la lettera maiuscola 'A' deve essere mappata consistentemente sulla stessa lettera minuscola.
5. **Casi Limite per Distanza di Editing**:
   - Distanza tra una parola e una stringa vuota (deve essere uguale alla lunghezza della parola).
   - Distanza tra due parole identiche (deve essere 0).
   - Parole che differiscono solo per una trasposizione (es. "torta" e "trota"), per verificare la corretta implementazione dell'algoritmo di Damerau-Levenshtein.