Laboratorio di Algoritmi e Strutture Dati

Relazione del Progetto: Gestione di un dizionario di parole e schemi (Go)

Introduzione

Il progetto "parole e schemi" gestisce un dizionario composto da **parole** (solo lettere minuscole) e **schemi** (che contengono almeno una lettera maiuscola). Il progetto permette la creazione di un **dizionario** che potrà essere popolato di parole e schemi attraverso operzioni bulk con file o dirette. Diverse operazioni sono permesse sugli elementi del dizionario. Il programma 41319A_valenti_alessandro.go contiene strutture, operazioni (e algoritmi) tutti in un unico file. Vengono poi sfruttati *formato_test.go, lib_test.go e utils_test.go* forniti dalla Prof. V. Lonati per eseguire test supplementari descrtitti in questo documento. In aggiunta è stato creato un file efficienza_test.go con tag=perf (per performace). Tutti i files del progetto incluso questo in formato markdown e pdf sono scricabili anche da https://github.com/sandroV1972/parole/

Descrizione di 41319A_valenti_alessandro.go

Risposta al problema della traccia

Il problema proposto ruota intorno ad una struttura *dizionario* che mantiene la Parole e gli Schemi. Oltre a normali operazioni sono richieste operazioni di risoluzione di catene e gruppi di parole. Per questo motivo è stata aggiunta sottostruttura GrafoCatena che viene aggiornato all'inserimento e alla cancellazione delle parole. Il GrafoCatena è una lista di adiacenza che mantiene per ogni parola del dizionario la lista delle parole di distanza 1 (secondo le specifiche del problema) Aggiorna Grafo. Il programma legge una serie di righe da **stdin** che contengono comandi definiti dal progetto che consentono di effettuare operazioni sul **dizionario**.

```
type dizionario struct {
    Parole
map[string]struct{}
    Schemi
map[string]struct{}
    GrafoCatena map[string]
[]string
}
```

Il dizionario contiene una mappa per le **Parole** una per gli **Schemi** e una mappa **GrafoCatena** che rappresenta la lista di connessioni a distanza 1 di ogni parola. Un percorso tra due parole nel **GrafoCatena** rappresenta una **catena**. Si è scelto di modellare Parole e Schemi con mappe perchè in Go, con *e elemento* della mappa (chiave), *trova(e)* in O(1) ammortizzato, *elimina(e)* in O(1) ammortizzato, la mappa mi garantisce che non vi siano duplicati. Il lato negativo di quuesta scelta è che la mappa non garantisce l'ordine quando si recuperano i dati per cui la **stampa** di *parole* e *schemi* richiede di copiare in una slice con **tempo e spazio O(n)** e poi ordinarle **tempo O(n log n)** quindi O(n+nlogn)=O(nlogn). In tutte le operazioni chiave (inserisci, elimina, ricerca, compatibilità) serve soprattutto test di appartenenza e aggiornamenti rapidi. Il GrafoCatena viene aggiornato a ogni inserimento di nuove parole nel dizionario. Ricavo quindi facilmente una catena(x, y) e un gruppo(x).

Grafo Catena

Il grafo modellato è un grafo non orientato (non pesato) con componenti connesse multiple (alcune parti possono non essere mutualmente ragiungibili). La struttura dati scelta è una lista di adiacenza implementata con una mappa di mappe. Ogni chiave è un nodo e il valore è una (mappa di [string]struct{}) che rappresenta la lista di adiacenza di vicini. La lista di adiacenza è una `map[string]struct{}` per permetterci ricerca e aggiornamenti in O(1).

Vale la pena soffermarsi sulla scelta implementativa sottolineando le differenze con un approccio progettuale diverso. L'aggiornamento del Grafo può essere fatto seguendo due approcci:

1. Scansione del dizionario Per ogni parola u di lunghezza L:
    - Confronti u con tutte le N parole del dizionario usando l'algoritmo di Damerau–Levenshtein (Levensthein 1965, Damerau 1964) in $O(L^2)$.
    - Costo per inserimento (o per ogni passo di BFS): $O(N \times L^2)$. Qui la complessità cresce linearmente con la dimensione del dizionario N e quadraticamente con la lunghezza L della parola.
2. Generazione on-the-fly dei vicini (genero tutti i possibili vicini senza cercarli nel dizionario) Per la stessa parola u di lunghezza L si generano circa:
    - L cancellazioni
    - L-1 trasposizioni
    - L×26 sostituzioni
    - L×26 inserzioni in $O(L \times |\Sigma|)$ operazioni (qui $|\Sigma| = 26$).
    - Per ciascuno dei ~2L + 52L candidati fai un lookup O(1) in map[string]struct{}.
    - Costo per inserimento (o per passo di BFS): $O(L \times |\Sigma|)$.

La complessità non dipende da N (la dimensione del dizionario) se non nel più che trascurabile fattore dei lookup O(1), ma cresce solo con L e con la dimensione dell'alfabeto.

Quando conviene quale?

- Se N è molto grande (milioni di parole) e L moderato (poche decine), l'approccio che genera vicini con DL è decisamente più veloce, perché $O(L \cdot |\Sigma|) \ll O(N \cdot L^2)$.
- Se L molto grande (centinaia/molti caratteri) ma N piccolo (pochi elementi), si potrebbe favorire la scansione del dizionario, ma in pratica con L ≤ 50 può essere grande, quindi "genera vicini" è quasi sempre preferibile.

In sintesi:

- Scansione → complessità $O(N \cdot L^2)$ per parola
- Generazione → complessità $O(L \cdot |\Sigma|)$ per parola

Per rispondere alla richiesta del problema di un entità dizionario unica (singleton), che può essere creata se non esistente, o resettata nei contenuti se già esistente, si crea una istanza in *main()* ma viene utilizzato sempre e solo un puntatore a quella istanza in tutti i metodi.

```
var d *dizionario
...
func main() {
    ...
    dict := newDizionario()
    d = &dict
    ...
}
```

Vengono quindi esguiti in sequenza i comandi inseriti in *stdin* fino a quando non viene inserito il comando **t**.

Crea

- Crea un nuovo dizionario se non esistente o ricrea le strutture del dizionario con nuove strutture vuote.
- La crezione della struttura richiede O(1).
- Popolare da file (*c nomefile.txt'*) richiede O(n) con n=numero di parole/schemi caricate nel dizionario.

Inserisci

- Inserisce una parola o schema nel dizionario in base alla presenza di lettere maiuscole nella stringa caricata. Questa operazione richiede di convertire in minuscole la parola che richiede O(L) con L lunghezza della parola e poi confrontarla con l'originale che richiede O(L) quindi con una complessità di O(2L) cioè O(L)
- L'inserimento richiede O(1)
- Verifica duplicati in tempo costante O(1)
- Ad ogni inserimento viene aggiornato il grafo del dizionario chiamando **aggiornaGrafo(w, ADD)**

  #### Aggiorna Grafo

  - Aggiungi parola:
    - Se la parola w non esiste in GrafoCatena del dizionario la aggiunge [O(1)]
    - Se la parola esiste calcola le possibili permutazioni di distanza 1 della parola w, esegue un lookup nel dizionario e se la permutazione esiste la aggiunge alla lista di adiacenza (vicini) di w in GrafoCatena [Vedi Grafo Catena]
  - Rimuovi parola:

- Chiamato con **aggiornaGrafo(w, REMOVE)**
- Elimina la parola dal dizionario in O(1)
- Elimina la chiave (parola) in GrafoCatena dopo aver rimosso la parola dalla lista di adiacenza di tutte le parola nella sua stessa lista di adiacenza. Il costo di questa operazione è O(n) con n lunghezza della lista di adiacenza della parola da eliminare [caso peggiore la parola dista 1 da tutte le altre parole del dizionario]

Elimina

- Rimuove la parola dal dizionario in O(1) grazie alla struttura dati scelta.
- Aggiorna il GrafoCatena vedi **aggiornaGrafo(w, REMOVE)**

Carica

- Legge da file parole e schemi. Richiama *inserisci* per ogni token del file. Ogni operazione viene eseguita in tempo costante O(1) per un tempo totale O(n) con n numero di parole/schemi nel file

Compatibile

- Confronta lettera per lettera parola e schema per verificare se esiste un'assegnazione coerente di lettere tra i due. Il confronto viene fatto lettera per lettera per posizione.
- Richiede un **tempo O(L)**, con L lunghezza dello schema/parola

Distanza

- Stampa la distanza tra due parole calcolata con **distDL(w1, w2)** che usa l'algoritmo di Damerau-Levensthein per calcolare la distanza tra due parole (usa inserimento, sostituzione, eliminazione e scambio calcolata con la matrice n x m con **O(n × m)**, dove n = len(w1), m = len(w2)).

Catena

- Utiizziamo un algoritmo BFS in quanto ho un albero non orientato non pesato (ogni collegamento ha peso 1) su parole collegate da distanza di editing 1 a partire dal GrafoCatena per calcolare la catena di parole tra le due parole **(w1, w2)**. BFS visita sempre i nodi piu vicini per cui appena trova *w2* possiamo assumenre quello sia il percorso più breve.
- Strutture ausiliarie sono mappa *visited* e *parents* per tenere traccia dei nodi visitati e dei nodi padre più una coda, quindi spazio O(3V) o O(V) con V nodi del grafo, per le tre strutture richieste. Viene creata una coda e si ricostruisce il percorso una volta trovata *w2* in tempo O(V+E) con V nodi e E archi del grafo.

Gruppo

- Secondo la definzione fornita nella traccia devo trovare tutte le connessioni a distanza 1 a partire da una singola parola **w**. Vuol dire visitare iterativamente tutte le liste di adiacenza

delle parole nella lista di adiacenza di *w*. Con una struttura ausiliaria coda si effettua una visita in ampiezza del grafo a partire dalla parole con un **tempo O(V+E)** dove V sono i nodi e E sono gli archi del grafo.
- Le strutture ausiliarie utilizzate sono la coda, e le mappe per i nodi visitati e il gruppo:

```
    g := make([]string, 0)
    visit :=
make(map[string]bool)
    queue := []string{w}
```

Anche qui lo **spazio** può essere calcolato in **O(V)**

Test del programma

Oltre ai testi forniti con il problema sono stati implementati altri test qui brevemente descritti. Utilizzando i file forniti con il progetto basta eseguire go test -v per una rassegna di tutti i test

- elimina una parola e uno schema inesistente (nessun output)
- inserire una parola o uno schema con caratteri non "a...z" o "A...Z" (nessun output)
- inserire una parola o uno schema duplicati (output senza duplicati, il secondo inserimento non avviene)
- ricerca di una catena vuota tra due parole che non hanno una serie di parole a distanza 1 tra di loro (output deve indicare 'non esiste')
- ricerca di un gruppo di una parola non in dizionario (output deve indicare 'non esiste')
- verifica che venga trovata la catena più breve nel grafo a---ac | / | aa acc | \ | aaa--aac
- ricerca di un gruppo a partire da una parola non nel dizionario (output 'non esiste')

Tests su input/elaborazioni molot grandi

- Caricare 100000 parole da file *mega_dizionario*
    - c mega_dizionario
    - p
    - t
- Stampare catene da >1000 nodi
    - c mega_dizionario
    - c a lajempkxlmnrohvrrdxpggmpsbtjqkkrnchxbzvvzfoatqbgap
    - t
- Stampare gruppi da >1000 nodi [g a]
    - c mega_dizionario
    - g a
    - t

Considerazioni finali

Scelte progettuali

Nel progetto abbiamo scelto di mantenere, nel Dizionario, una struttura di grafo (lista di adiacenza) aggiornata all’**inserimento** di ogni nuova parola. Questo sposta parte del costo computazionale dall’atto di eseguire una catena o un gruppo (query) al momento dell’inserisci. In particolare: • All’inserimento, il costo ammortizzato è $O(L \cdot |\Sigma|)$, con L = lunghezza della parola e $|\Sigma| = 26$ (generazione on-the-fly dei vicini). • Le operazioni di catena e gruppo diventano semplici BFS su un grafo aggiornato, con complessità $O(V+E)$ in tempo e $O(V)$ in spazio (V = numero di parole, E = numero di archi). Questa scelta è vantaggiosa in scenari con molte ricerche e pochi inserimenti, garantendo risposte rapide per catene e gruppi di decine o centinaia di migliaia di nodi. I costi in tempo e spazio rimangono sempre **lineari**.

Test formato e stress test

Utilizzando i test forniti con il progetto sono stati inseriti altri e significativi test per il formato degli inserimenti e delle richieste (casi limite e patologici). E' stato creato un test file dizionario con 100000 parole con una catena da 1000 e un gruppo da 2000 parole massimo di 50 caratteri per testare il caricamento e la gestione dell'aggiornamento del Grafo Catena che sembra non subire significativi rallentamenti.

Limiti e possibili miglioramenti

1. Non avendo avuto indicazioni su limiti e uso del progetto si è spostato il peso sulla ricerca e meno sull'inserimento. Abbiamo ipotizzato limiti sia alle dimensioni del dizionario sia alla dimensione delle parole.
2. Nei limiti delle richieste progettuali si pensa che le strutture e gli algoritmi implementati siano ottimali in termini di tempo e spazio.