Progetto “Parole”- Relazione

Matricola: 13375A
Studente: Luigi Lonigro

INTRODUZIONE
Il progetto sviluppato ha come obiettivo la gestione di un *dizionario* contenente *parole* e *schemi*, e l'esecuzione di operazioni su di essi, come la *ricerca*, il calcolo della *distanza di editing*, la generazione di *catene* di *parole* e l'analisi di *compatibilità schema-parola*.

Il programma è scritto in linguaggio Go ed esegue operazioni leggendo comandi dallo standard input (stdin) e producendo output formattati su standard output, come richiesto dalla traccia.

MODELLAZIONE E SCELTE PROGETTUALI

Il dizionario è rappresentato dal tipo `dizionario`, definito come una struttura contenente:

- `parole map[string]bool`: insieme delle parole memorizzate.
- `schemi map[string]bool`: insieme degli schemi memorizzati.
- `grafoP map[string][]string`: grafo delle parole simili

La scelta di utilizzare *mappe* al posto di *slice* per memorizzare *parole* e *schemi* è data dalla necessità di fare cambiamenti nel dizionario (come le funzioni `inserisci(w)` ed `elimina(w)`) poiché queste ultime permettono operazioni come ricerca, inserimento e cancellazioni con una velocità maggiore rispetto alle slice (la mappa ha O(1) invece di una ricerca lineare O(n) di una slice).
In aggiunta a quanto detto precedentemente, le mappe permettono anche un veloce controllo di presenza per parole e schemi nel dizionario.
(per le parole in particolare questo è utile per funzioni come `catena(x, y)`)

La distinzione tra parola e schema è basata sulla presenza di lettere maiuscole (`isParola()`) restituisce `false` in tal caso).

Il grafo `grafoP` rappresenta la rete di parole connesse da una distanza di editing pari a 1, secondo la definizione della traccia. Viene costruito solo al primo utilizzo della funzione `catena` per evitare elaborazioni non necessarie. Quando il dizionario viene modificato, tramite inserimenti e rimozioni, il grafo viene azzerato e dev’essere nuovamente inizializzato così da evitare errori.

SCELTE IMPLEMENTATIVE

Comandi

L'input viene gestito nella funzione `main()`, dove si legge riga per riga e si invoca `esegui(d, riga)`.
La funzione `esegui()` decodifica il comando (ad esempio `i aB`, `r aC`) e chiama le corrispondenti operazioni sul dizionario.

Funzionalità implementate

- **Inserimento/eliminazione**: `inserisci()` e `elimina()` operano su parole o schemi usando `isParola()` per discriminare.
- **Stampa**: `stampa_parole()` e `stampa_schemi()` stampano i rispettivi insiemi nel formato richiesto (una parola/schema per riga tra parentesi quadre).
- **Ricerca compatibilità schema-parola**: `ricerca(schema)` itera su tutte le parole e verifica la compatibilità tramite `compatibileConSchema()`.
- **Distanza di editing**: `distanza()` implementa la distanza di Damerau-Levenshtein (inserimento, cancellazione, sostituzione, scambio).
- **Catena di parole**: `catena(x, y)` costruisce un grafo delle parole simili ed esegue una BFS per trovare la sequenza più breve da `x` a `y`.

COMPATIBILITA' SCHEMA (`compatibileConSchema()`)

```go
func compatibileConSchema(schema, parola string) bool {
    if len(schema) != len(parola) {
        return false
    }
    mapAssegnazioni := make(map[byte]byte)

    for i := 0; i < len(schema); i++ {
        s := schema[i]
        p := parola[i]

        if s >= 'A' && s <= 'Z' {
            if val, assegnata := mapAssegnazioni[s]; assegnata {
                if val != p {
                    return false
                }
            } else {
                mapAssegnazioni[s] = p
            }
        } else {
            if s != p {
                return false
            }
        }
    }
    return true
}
```

usa una mappa di assegnazioni (`mapAssegnazioni`) per salvare la lettera maiuscola alla lettera minuscola scelta.

la parola non è compatibile se:

- **non ha la stessa lunghezza dello schema**
- **non ha le stesse minuscole dello schema**
- **ha troppe minuscole da assegnare alle maiuscole dello schema**

DISTANZA DI EDITING (`distanza()`) (AUSILIARIO: `Min(a, b)`)

Da una prima lettura superficiale, **la distanza di editing** in questo progetto poteva essere implementata sia con **DL (Damerau-Levenshtein)** sia con **OSA (Optimal String Alignment)**, i motivi che ci portano a scegliere **DL** sono i seguenti:

- **Nessuna restrizione sulle trasposizioni:** non abbiamo limiti nel numero di trasposizioni che possiamo eseguire, questo esclude OSA visto che quest'ultima non supporta trasposizioni ripetute o sovrapposte

- **Accuratezza:** DL è più accurata e più realistica, OSA potrebbe risultare in una distanza incompleta, che, soprattutto in `catena(x, y)`, produrrebbe risultati errati

CATENA (`catena(x, y)`) (AUSILIARIO: `d.costruisciGrafo()`)

```go
func (d *dizionario) catena(x, y string) {
    if _, ok := d.parole[x]; !ok {
        fmt.Println("non esiste")
        return
    }

    if _, ok := d.parole[y]; !ok {
        fmt.Println("non esiste")
        return
    }
    if d.grafoP == nil || len(d.grafoP) == 0 {
        d.costruisciGrafo()
    }

    listaDiEsplorazione := []string{x}
    nodiVisitati := map[string]bool{x: true}
    predecessori := map[string]string{}

    for len(listaDiEsplorazione) > 0 {
        nodoCorrente := listaDiEsplorazione[0]
        listaDiEsplorazione = listaDiEsplorazione[1:]

        if nodoCorrente == y {
            break
        }

        for _, vicini := range d.grafoP[nodoCorrente] {
            if !nodiVisitati[vicini] {
                nodiVisitati[vicini] = true
                predecessori[vicini] = nodoCorrente
                listaDiEsplorazione = append(listaDiEsplorazione, vicini)
            }
        }
    }

    if !nodiVisitati[y] {
        fmt.Println("non esiste")
        return
    }

    catena := []string{}
    for nodo := y; nodo != ""; nodo = predecessori[nodo] {
        catena = append([]string{nodo}, catena...)
    }

    fmt.Println("(")
    for i := 0; i < len(catena); i++ {
        fmt.Println(catena[i])
    }
    fmt.Println(")")
}
```

La funzione implementa una **BFS (Breadth-First Search)**

Viene costruito il grafo quando la funzione viene chiamata la prima volta **(con velocità O(n^2))**

BFS non solo trova il percorso più corto da x a y come da richiesta, ma è anche molto efficiente quando ci sono tante parole **(O(V + E)).**

Vengono quindi create:
Una lista di esplorazione,
una mappa di nodi visitati,
una mappa di nodi e dei loro predecessori.

nel nostro caso logicamente i nodi sono le parole e sono collegati l'uno all'altro se la distanza di edit è uguale ad 1

Il resto dell'implementazione esplora i nodi partendo dalla testa della lista di esplorazione, per ogni nodo non visitato aggiorna la lista di esplorazione aggiungendolo, aggiorna i predecessori e la lista dei nodi visitati.

Trovata la fine del cammino ripercorriamo la strada partendo dalla fine (da y) ed usando la mappa dei predecessori torniamo all'inizio del cammino (x)

TESTS AGGIUNTIVI

```
Input:
c
i abba
i cddc
r ABBA
t
Output:
ABBA:[
abba
cddc
]
Motivazione: test per parole palindrome di 4 lettere
```

TEST PER LETTERE PALINDROME

```
Input:
c
d casa casa
t
Output:
0
Motivazione: la distanza tra due parole identiche deve essere 0
```

TEST PER DISTANZA 0 CON PAROLE IDENTICHE

```
Input:
c
i aaba
i abba
i adda
r aBBA
t
Output:
aBBA:[
abba
adda
]
Motivazione: test di schema con lettere ripetute e fisse
```

TEST PER SCHEMA CON LETTERE RIPETUTE E FISSE

(Serve a dimostrare che l'assegnazione delle lettere maiuscole di uno schema viene mantenuta solo per controllare la compatibilità di una parola con quest'ultimo, con la parola successiva si potrà assegnare lettere maiuscole ad altre lettere minuscole)

```
Input:
c
i casa
i topo
c casa topo
t
Output:
non esiste
Motivazione: le parole non hanno distanza di editing pari a 1 né sono collegate
```

TEST PAROLE CATENA TROPPO DISTANTI

```
Input:
c
i aa
i aba
i abb
c aa abb
e aba
c aa abb
i aba
c aa abb

Output:
(
aa
aba
abb
)
non esiste
(
aa
aba
abb
)
Motivazione: verifica che al cambiare del dizionario anche il grafo cambia
```

TEST MODIFICA DIZIONARIO MODIFICA GRAFO

(Serve per verificare che quando il dizionario viene aggiornato, aggiungendo o rimuovendo una parola, quindi un nodo, anche il grafo verrà modificato per non produrre errori dati da parole che potrebbero non far più parte del dizionario)