Relazione progetto d'esame di algoritmi e strutture dati

Per rappresentare il piano è stato implementato un grafo non orientato pesato e fortemente connesso. Il grafo è implementato mediante una lista, ad ogni inserimento viene aggiornata, con l'inserimento in testa di un'automa e in coda di un ostacolo.

Strutture dati coinvolte e tipi dato principali

Il tipo `piano` è usato per rappresentare un grafo non orientato pesato e connesso, mediante lista doppiamente concatenata, contenete l'elenco degli automi e delgi ostacoli. Ogni ostacolo o automa è memorizzato tramite una struttura `punto` che rappresenta i nodi del grafo. Il `piano` è composto dai campi `inizio` e `fine` che sono puntatori che puntano rispettivamente al `punto` che sta all'inizio della lista e a quello alla fine della lista. Gli automi staranno nella prima metà della lista e gli ostacoli nella seconda metà, questa scelta è stata presa per rendere più "comoda" e rapida la ricerca all'interno del campo.

Il tipo `punto` è usato per rappresentare un generico punto nel piano, e un generico nodo della lista. Esso è commposto da un campo `id`, due valori interi che rappresentano rispettivamente la coordinata x e la coordinata y, un campo `richiamo` di tipo bool, che indica se il richiamo α vale per l'automa considerato, e due puntatori ad altri punti: `successivo` e `precedente`, che sono rispettivamente il nodo successivo e il nodo precedente del punto considerato nella lista. Essendo anche gli ostacoli punti nel piano, contengono la sezione `richiamo`, che risulta inutile per loro. Questa scelta è stata fatta al fine di non dover implementare un altro tipo di dato per l'ostacolo e poterli contenere nella stessa struttura dati degli automi, come punti all'interno del piano.

Il piano nel programma è rappresentato come una variabile globale `Campo` di tipo `piano`. Ho fatto in questo modo per rendere più semplici le operazioni all'interno del programma, da implementare, e non dover necessariamente passare come parametro il piano ogni volta che bisogna fare un'operazione su un punto interno ad esso.

Metodi e funzioni

Funzioni

- `esegui(p piano, s string)` : Questa funzione prende in input un piano `p` e una stringa `s` e, in base al contenuto di `s` vengono eseguite operazioni diverse sul piano.
- `newPiano() piano` : Questa funzione viene invocata ogni volta che bisogna creare un nuovo piano. Viene invocata dalla lettera `c` all'interno della stringa `s`. Vine associato una nuova variabile di tipo `piano` alla variabile globale `Campo`. Questa funzione implementa l'operazione `crea()`.
- `stampa()` : Questa funzione stampa prima un elenco contenente tutti gli automi che fanno parte del piano e poi un altro contenente tutti gli ostacoli. Questa funzione stampe sempre il piano secondo il formato di output richiesto.
- `stato(x, y int)` : Questa funzione stampa il tipo di entità presente in un determinato punto. Se nelle coordinate `x`, `y` è presente un automa, allora la funzione stamperà `A`, se vi è un punto fancente parte di un ostacolo, stamperà `O` e invece se non c'è niente, stamperà `E`.
- `posizioni(alpha string)` : Questa funzione stampa gli automi che hanno come prefisso `alpha`. Questa funzione stampa gli automi nel formato output richiesto.
- `automa(x, y int, eta string)` : Questa funzione aggiunge un automa al campo se il punto (x, y) non fanno parte di nessun ostacolo, altrimenti non fa nulla. Se le coordinate (x, y) non fanno parte di un'ostacolo allora controlla se l'automa `eta` esite già e in caso affermativo sposta l'automa di nome `eta` nella posizione (x, y), altrimenti lo crea nuovo.

- `ostacolo(x0, y0, x1, y1 int)` : Questa funzione aggiunge un ostacolo all'interno del `piano`. Se l'area del quadrato compresa tra il vertice in basso a sinistra, `x0` e `y0`, e il vertice in alto a destra, `x1` e `y1`, contiene un'automa la funzione termina. Altrimenti aggiunge un punto con coordinate, `(x0, y1)` con id nel segiente formato: `(x0, y0, x1, y1) ostacolo`.

  > **Considerazioni sul comportamento dell'entità ostacolo:**
  >
  > Per il calcolo dell'area del quadrato verranno usate le funzioni `dentroAreaOstacolo` e `estraiCoordinate` all'occorrenza all'inetrno del programma. Ho scelto di implementare in questo modo il comportamento e la rappresentazione nel progetto perchè mi sembrava il modo più efficienete per verificare se due coordinate x, y facessero parte dell'area di un ostacolo. In questo modo ho potuto mantenere un unico tipo di dato per automi e ostacoli.
  >
  > - `dentroAreaOstacolo(x, y int) bool` : Questa funzione serve per calcolare se un punto è all'interno dell'area di un ostacolo. Restituisce `true` se il punto `(x, y)` fa parte dell'area di un generico ostacolo all'interno del campo.
  >
  > - `estraiCoordinate(id string) (x0 int, y0 int, x1 int, y1 int)` : Questa funzione è ustata per estrarre le coordinare del punto in basso a sinistra e in alto a destra dall'id. Esso infatti è formattato in modo che vi siano le coordina dei due punti e la dicitura `ostacolo` subito dopo, in qusto modo: `Ostacolo.id = "x0,y0,x1,y1,ostacolo"`.

- `richiamo(x, y int, alpha string)` : Questa funzione avvia il richiamo per, e soltanto, tutti gli automi che come prefisso hanno la stringa `alpha`. Setta il campo `richiamo` degli automi interessati a `true`. Inoltre alloca una variabile globale `Sorgente` che sarà il punto che gli automi con prefisso `alpha` dovranno raggiungere.

- `esistePercorso(x, y int, eta string)` : Questa funzione è usata per far stampare a schermo se esista o meno un percorso di distanza $D(P(\eta), (x, y))$ che in caso affermetivo stampa `SI` altrimenti `NO`. Questa funzione stampa `NO` anche se il punto (x, y) fa parte dell'area di un ostacolo.

  > **Considerazione sulla funzione `esistePercorso`**
  >
  > L'implementazione della funzione delega alcune operazioni su altre due funzioni: `avanza` e `calcolaDistanza`. Questa scelta è stata presa per *appesantire* il meno possibile il codice della funzione principale e per poter dividere concettualmente il problema della ricerca del percorso. Di fatto questa scelta non è stata presa per fini legati alla performance del programma ma ai fini di rendere più facile implementare la risoluzione del problema della ricerca del percorso.
  >
  > - `avanza(p *punto, passi int) (*punto, int)` : Questa è una funzione ricorsiva che deve cercare il percorso minore per arrivare al punto dove si trova `Sorgente`. Come caso base, la funzione richiede `passi == 0`, `passi` è la distanza minima $D(P(\eta), (x, y))$. Il passo ricorsivo controlla dove il punto `p` si avvicina di più alla sorgente del segnale. Controlla poi su quale asse del piano, relativo alle coordinate del punto, ci sono più ostacoli. Per fare ciò mi sono appoggiato ai metodi `(p *punto) presenzaOstacoloPercorsoX(y int)` e `(p *punto) presenzaOstacoloPercorsoY(x int)` (questi metodi sono discussi nella sezione relativa ai metodi). Se non ci sono percorsi, che riducano la distanza dal punto `p` alla sorgente e che non passino attraveso un ostacolo, il punto `p` va in *stallo* mentre i `passi` diminuiscono.
  >
  > - `calcolaDistanza(x0, y0, x1, y1 int) int` : Questa funzione calcola, e restituisce, la distanza tra il punto $(x0, y0)$ e il punto $(x1, y1)$.

Metodi

- `(*piano)cerca(x, y int, id string)*punto` : Questo metodo serve per cercare uno specifico punto, attraverso le coordinate ed eventualmente il nome, all'interno del piano. Se il punto non è presente il metodo restituisce `nil`, altrimenti da il punto all'ineterno del piano.
- `(p *punto) presenzaOstacoloPercorsoX(y int) bool` :

- `(p *punto) presenzaOstacoloPercorsoY(x int) bool` :

Esempi di esecuzione e casi limite