

# Automi e Segnali

---

## 1. Organizzazione del Programma

Il programma è organizzato su 5 file `.go`, tutti nello stesso package `main`.

### Struttura del Progetto

I 5 file presenti nel progetto sono:

- `main.go`: Contiene il punto di ingresso del programma.
- `piano.go`: Definisce la struttura del piano e l'algoritmo di ricerca di percorso minimo.
- `automa.go`: Contiene alcuni metodi relativi agli automi sul piano.
- `ostacolo.go`: Contiene la struttura relativa agli ostacoli e alcuni metodi relativi a quest'ultimo sul piano.
- `utils.go`: Contiene funzioni di utilità e alcune strutture dati utilizzate.

### Compilazione

Per compilare il programma, eseguire il seguente comando nella directory del progetto in cui sono presenti i file sorgente `.go`:

```
go mod init nomefile
go mod tidy
go build
```

## 2. Descrizione del Programma

### Sintesi del Programma

Il progetto implementa un sistema per creare e gestire un piano cartesiano con automi e ostacoli. Gli automi possono essere aggiunti, stampati, ricercati e richiamati da un segnale. Gli ostacoli possono essere aggiunti al piano e stampati. Il programma include anche la funzionalità per trovare percorsi tra punti specifici, detti segnali, e automi con un certo prefisso utilizzando un algoritmo di ricerca del percorso minimo.

### Comandi del programma

- Il programma legge i comandi dall'input standard e li esegue tramite la funzione `esegui(p piano, s string)`
  - `c`: Crea un nuovo piano vuoto, se ne esiste già uno, lo sovrascrive.
  - `s <x> <y>`: Mostra lo stato del piano nel punto di coordinate (x, y).
  - `S`: Stampa l'elenco degli automi e successivamente quello degli ostacoli.

- **a <x> <y> <η>**: Se nella posizione (x, y) non è presente un ostacolo, inserisce un automa di nome η nel punto (x, y).
- **o <x0> <y0> <x1> <y1>**: Se nell'area da (x0, y0) a (x1, y1) non è presente nessun automa, inserisce un ostacolo in quell'area.
- **r <x> <y> <α>**: Emette un **segnale** da (x, y) con prefisso α, se l'automa con prefisso α è raggiungibile, sposta l'automa nel punto del segnale, così per ogni automa con quel prefisso.
- **p <α>**: Stampa le posizioni degli automi con prefisso α.
- **e <x> <y> <η>**: Verifica se un automa di nome η può raggiungere il punto (x, y) in distanza minima.
- **f**: Termina il programma.

## Scelte Implementative

### • Strutture Dati Utilizzate

- Punto{x, y int} struttura che rappresenta un punto bidimensionale con due coordinate rappresentate da numeri interi
- Ostacolo{x0, y0, x1, y1 int} struttura che rappresenta un ostacolo con la coppia di interi (x0, y0) che rappresenta l'angolo in basso a sinistra e la coppia di interi (x1, y1) che rappresenta l'angolo in alto a destra.
- ostacolo{ost Ostacolo, next \*ostacolo} struttura come nodo di linkedList.
- ostacoli{\*ostacolo} struttura di tipo linkedList composta da ostacoli, presenta un metodo **insert(...)** che permette di inserire un ostacolo in testa alla lista degli ostacoli.
- Piano{automi map[string]Punto, \*ostacoli} struttura che rappresenta il piano attraverso: automi, una mappa che associa ad ogni nome di automa, che è univoco, la sua posizione come un Punto; ostacoli, che contiene tutti gli ostacoli sul piano.
- PuntoHeap []Punto è una coda con funzioni di **push(...)**, **pop(...)**, **Less(...)**, **Swap(...)** **Len(...)**. Questa struttura permette all'algoritmo di ricerca di essere più efficiente e diminuire i tempi di accesso.

### • Implementazione degli Algoritmi

- La ricerca è stata implementata attraverso l'uso di un algoritmo A\*. Tramite l'uso di mappe per tenere conto dei percorsi di minor lunghezza l'algoritmo è in grado di trovare il cammino minimo tra due punti, se esistente, uguale alla distanza di Manhattan. Gli algoritmi di tipo A\* sono algoritmi che permettono di trovare sicuramente il percorso minimo (se esiste) tra due punti e hanno un'efficienza ottima. Viene utilizzato un heap secondo le specifiche della libreria '**container/heap**', questa struttura permette un tempo **O(logn)**, dove **n** è **h.Len()**, ovvero il numero di elementi nell'heap, per l'inserimento e il pop degli elementi. L'algoritmo può anche essere implementato in maniera bidirezionale ma potrebbe sia aumentare che diminuire il costo in termini di spazio e aumentare la complessità generale dell'algoritmo.
- La ricerca dei vicini ad un punto è fatta tramite il controllo dei punti con distanza 1 dal punto di cui si cercano i vicini, se la distanza tra il punto trovato e il goal è minore e il punto non appartiene ad un ostacolo, viene inserito nell'array dei vicini, nel caso una di queste condizioni siano false non succede niente.

- La funzione `stato(x, y)` itera tra tutti gli automi e tra tutti gli ostacoli per determinare se si trovano in quel punto attraverso un controllo delle coordinate.

- **Scelte Scartate**

- Algoritmo di tipo Best-First-Search per la ricerca di un cammino minimo. l'algoritmo permetteva di trovare in modo iterativo il percorso tra la sorgente e un automa ma in alcuni casi veniva resituito un percorso che non fosse minimo nonostante ne fosse presente un altro minimo.
- Rappresentazione degli automi come struttura `Automa(Punto{a, b int}, nome string)`, scartata poiché non permette una buona ottimizzazione del codice, diversamente dall'uso delle mappe.
- Uso di una mappa `posizioni[Punto]string` che permetteva di assegnare ad ogni punto un carattere per permettere una più veloce esecuzione della funzione `stato`, avrebbe permesso un tempo di accesso  **$O(1)$** , permettendo una velocità maggiore della funzione `stato()` e nella ricerca dei vicini ma nel caso di ostacoli molto grandi non sarebbe stato possibile l'inserimento di questi, dato che avrebbe superato le dimensioni della memoria centrale.

## Calcoli relativi ai costi delle Operazioni

- **Costo in Tempo**

- Controllo dello stato in un punto:  **$O(a+o)$** , dove  **$a$**  è il numero di automi e  **$o$**  il numero di ostacoli.
- Controllo presenza automa prima di inserire un ostacolo:  **$O(a)$**  dove  **$a$**  è il numero di automi nel piano.
- Controllo presenza di un ostacolo prima di inserire un automa: usa `stato(...)`, quindi  **$O(a+o)$** , dove  **$a$**  è il numero di automi e  **$o$**  il numero di ostacoli.
- Ricerca del percorso: caso peggiore  **$O(n(a+o)+n*\log n)$**  perché nel peggiore dei casi il `for` itera su tutti i punti nella griglia, richiedendo tempo  **$O(n)$**  e ogni operazione di inserimento o rimozione dall'heap richiede tempo  **$O(\log n)$**  è anche necessario tenere conto del calcolo dei vicini, che usa la funzione `stato(...)`, dato che in caso uno dei vicini sia un ostacolo o un automa, non viene inserito nella slice dei vicini.
- Ricerca automi con un prefisso specifico alpha:  **$O(a*p)$**  dove  **$a$**  è il numero di automi nel piano e  **$p$**  è la lunghezza in caratteri del prefisso.
- Richiamo degli automi: il ciclo `for` itera su tutti gli automi  **$O(a*c)$**  con  **$a$**  numero di automi e  **$c$**  è il numero di caratteri del prefisso. Dopodichè su tutti quelli che hanno il prefisso alpha controlla se esiste un percorso usando la funzione `esistePercorso(...)`, che ha costo  **$O(n*(a+o)+n*\log n)$** .

- **Costo in Spazio**

- Memorizzazione del piano:  **$O(o+a)$**  dove  **$o$**  è il numero degli ostacoli del piano  **$a$**  è il numero di automi nel piano.
- Memorizzazione degli ostacoli:  **$O(o)$**  dove  **$o$**  è il numero di ostacoli presenti sul piano.
- Memorizzazione degli automi:  **$O(a)$**  dove  **$a$**  è il numero di automi presenti sul piano.
- Memorizzazione della coda:  **$O(n)$**  dove  **$n$**  è il numero di punti considerati durante la ricerca del percorso.
- Memorizzazione delle mappe per la ricerca del percorso:  **$O(n)$**  dove  **$n$**  è il numero di punti considerati durante la ricerca del percorso.

## Esempi di esecuzione

## Esempio 1

Con input:

```
c
a 1 1 1
a 3 3 10
o 2 0 4 3
o 5 5 7 8
o -2 -4 -1 -1
s 1 1
s 3 1
s 7 8
s 10 10
S
f
```

abbiamo output:

```
A
E
O
E
(
1: 1,1
10: 3,3
)
[
(5,5)(7,8)
(-2-4)(-1,-1)
]
```

## Esempio 2

Con input:

```
c
a 1 2 1
e 10 2 1
o 2 0 4 3
e 10 2 1
e -10 2 1
o -4 0 -2 3
e -10 2 1
f
```

abbiamo output:

SI  
NO  
SI  
NO