Relazione del progetto d'esame di Algoritmi

Francesco Emanuele Corrado 29407A

a.a. 2024/2025

Figure 1: Unimi

Laboratorio di Algoritmi e Strutture Dati

Indice

Struttura Progetto

Il progetto si compone di un unico file go `29407A_corrado_francesco.go`, questa relazione ed una cartella contenente alcuni dei test effettuati. Per eseguire i file di test si può usare il seguente formato, dopo aver creato una cartella apposita per il file `29407A_corrado_francesco.go`

```
go mod init solution
go build
./solution < ./tests/29407A_corrado_francesco_test*numeroTest*IN
```

Descrizione Progetto

Il progetto permette di gestire un piano cartesiano sul quale vengono posizionati automi ed ostacoli secondo specifiche regole (es. gli automi hanno il nome formato da {0,1} e non posso creare un ostacolo sopra ad un automa). I comandi sono dati tramite righe di input con i comandi nel seguente formato:

- `c` : crea un piano vuoto ed elimina eventualmente quello già esistente
- `S` : stampa l'elenco di automi seguito dall'elenco di ostacoli contenuti al momento nel piano
- `s x y` : stampa lo stato del piano nel punto di coordinate `(x,y)`: A se è un automa, O se è un ostacolo E se non c'è niente

- `a x y w` : crea un automa di nome `w` (in {0,1}) alle coordinate `(x,y)` o se esiste già lo sposta alle coordinate
- `o x0 y0 x1 y1` : crea un ostacolo che va dal vertice `(x0,y0)` a `(x1,y1)`, se non sono presenti automi nell'area
- `r x y w` : crea un segnale di richiamo nel punto `(x,y)`. Tutti gli automi di prefisso `w` verificano se possono raggiungere il punto con distanza minima senza ostacoli e se possono si spostano in `(x,y)`
- `p w` : stampa la posizione di tutti gli automi di prefisso `w`
- `e x y w` : determina se esiste un percorso minimo senza ostacoli da `(x,y)` all'automa `w`
- `t x y w` : determina la tortuosità, ovvero il numero di cambi di direzione, minima tra il punto `(x,y)` e l'automa `w`
- `f` : termina l'esecuzione

Scelte Di Progettazione

Le scelte di progettazione del progetto sono state scelte cercando di rendere il progetto più ottimizzato, efficiente e veloce possibile. Alcune delle scelte utilizzate sono le seguenti:

Strutture Dati

- Automi, Occupati, Posizione Automi: sono stati implementati utilizzando delle mappe, in quanto le mappe sono perfette per gestire velocemente ed efficacemente con bassa complessità gli elementi di insiemi come questi, che rappresentano:
  - Automi: l'elenco di automi nel piano
  - Occupati: l'elenco di punti del piano occupati da ostacoli
  - Posizione Automi: l'elenco di di punti del piano che contengono automi
- Ostacoli, PriorityQueue: sono stati implementati con una slice in quanto sono efficienti per l'accesso sequenziale e permettono di aggiungere nuovi elementi in modo dinamico. In particolare i due rappresentano:
  - Ostacoli: una lista di ostacoli formati dai due punti di vertice di essi
  - PriorityQueue: una lista di elementi contenenti il nome di un automa e la sua distanza, in modo tale da poterli ordinare in questa coda a priorità
- Trienode: rappresenta un albero con tutti i nomi degli automi del piano. Il Trie (albero di prefissi) è una struttura dati specializzata per la ricerca di stringhe con prefissi comuni. È particolarmente efficiente per operazioni di ricerca di prefissi, come richiesto dalla funzione `richiamo`. Questo lo rende ideale per gestire i nomi degli automi e rispondere rapidamente ai segnali di richiamo basati sui prefissi.
- Piano: è una struttura che contiene al suo interno Automi, Ostacoli, Occupati, Posizione Automi e Trienode così da poter avere un quadro completo della situazione attuale del piano in qualsiasi momento e con accesso rapido
- Strutture di supporto: sono state utilizzate diverse strutture per supportare gli algorirmi e i ragionamenti utilizzati. Queste strutture aiutano a memorizzare dati ed effettuare operazioni che saranno poi descritti nelle Specifiche Implementative. Alcune sono:
  - stateRoad: il percorso di un automa
  - orderedRoad: percorso di un automa con indice di priorità
  - visitedNodes: i nodi visitati
  - DistanceRoad: un percorso con una distanza associata

Algoritmi

Trattandosi di una ricerca di percorsi su un piano cartesiano, ho ritenuto utilizzare una ricerca BFS (Breadth-First Search) la scelta migliore in quanto:

- Garantisce di trovare il percorso più breve
- Un piano cartesiano può facilmente essere descritto come una griglia dove ogni punto del piano è un nodo e ogni arco è un movimento possibile

- Una ricerca BFS facilità il calcolo della tortuosità in quanto essa registra i nodi precedenti rendendo semplice calcolare quanti cambi di direzione sono stati effettuati
- È efficiente

Scelte Implementative

Strutture Dati

Le scelte implementative del progetto sono state scelto cercando di rappresentare al meglio le scelte di progettazione rendendo il codice efficiente

- Punto: un punto nel piano, rappresentato da `x, y int`
- Automa: formato da un `Punto`
- Ostacolo: formato da due `Punto`
- Automi : gli automi sono stati salvati tramite una `map[string]Automa` dove `Automa` contiene un punto di coordinate `(x,y)` ovvero le coordinate dell'automa
- Ostacoli: gli ostacoli sono stati salvati in una lista `[]Ostacolo` dove `Ostacolo` contiene due punti con due coordinate `(x,y)` ovvero il vertice inferiore sinistro e il vertice superiore destro dell'ostacolo
- Occupati: I punti occupati da ostacoli sono gestiti grazie ad una mappa `map[Punto]bool` dove `Punto` è un punto nel piano e `bool` indica se quel punto è occupato da un ostacolo oppure no
- Posizione Automi: La posizione di tutti gli automi nel piano è gestita grazie ad una mappa `map[Punto][]string` dove `Punto` è un punto del piano e `[]string` è una lista di stringhe ovvero i nomi degli automi
- PriorityQueue: è stata implementata usando una lista come coda di priorià con le funzioni `Push Pop Less Swap` e `Len` utilizzando la libreria di go `containers/heap`
- Trienode: è stato implementato utilizzando una struttura `Trienode` con le relative funzioni `insert searchPrefix` e `collectAll`

Algoritmi

Spiegazione dettagliata del funzionamento delle funzioni principali

possibleRoads

La funzione possibleRoads calcola i possibili percorsi di un automa da un punto a un altro. Restituisce una lista di percorsi disponibili

Nel dettaglio:

La funzione è un metodo del tipo `piano` che prende in input un percorso `s`, le coordinate di destinazione `(x,y)` e una distanza `dist`

totalMoves definisce i movimenti possibili (su, destra, giù, sinistra) con le relative coordinate e direzioni.

```
func (p piano) possibleRoads(s stateRoad, x, y, dist int) []stateRoad {
totalMoves := []struct {
        ox, oy, d int
    }{
        {0, 1, 2}, // up
        {1, 0, 1}, // right
        {0, -1, 0}, // down
        {-1, 0, 3}, // left
    }
```

Vengono calcolati tutti i percorsi validi:

```
var moves []orderedRoad
for _, move := range totalMoves {
    ox := s.x + move.ox
    oy := s.y + move.oy
    if p.isOstacolo(ox, oy) {
        continue
    }
    if manhattanDist(Punto{ox, oy}, Punto{x, y}) != dist-(s.steps + 1) {
        continue
    }
    priority := -manhattanDist(Punto{ox, oy}, Punto{x, y})
    moves = append(moves, orderedRoad{move.ox, move.oy, move.d, priority})
}
```

- Per ogni movimento possible secondo `totalMoves`, calcola le nuove coordinate `(ox,oy)`
- Verifica se il nuovo punto è un ostacolo grazie a `isOstacolo` che restituisce true se nelle cordinate passate è presente un ostacolo. Se lo è, salta al prossimo movimento
- Verifica se la distanza di Manhattan dal nuovo punto alla destinazione è corretta, calcolandola tramita `manhattanDist`. La distanza Manhattan di due punti è calcolata sommando il valore assoluto della differenza delle coodinate di essi. Se non lo è, salta al prossimo movimento
- Calcola la priorità del movimento come l'opposto della distanza di Manhattan dal nuovo punto alla destinazione
- Aggiunge il movimento valido alla lista moves

Ordina la slice contenente tutti i percorsi validi ordinandoli per priorità tramite `sort.Slice`

```
sort.Slice(moves, func(i, j int) bool {
    return moves[i].priority < moves[j].priority
})
```

Viene creata una slice contenente i percorsi creati con i dati precedenti nel tipo `stateRoad`

```
var roads []stateRoad
for _, move := range moves {
    nx := s.x + move.ox
    ny := s.y + move.oy
    turns := s.turns
    if s.dir != -1 && s.dir != move.odir {
        turns++
    }
    road := stateRoad{x: nx, y: ny, steps: s.steps + 1, dir: move.odir, turns: turns}
    roads = append(roads, road)
}
return roads
```

- Per ogni movimento, calcola le nuove coordinate `(nx, ny)`
- Calcola il numero di cambi di direzione `turns`. Se la direzione del movimento corrente è diversa dalla direzione precedente, incrementa il numero di cambi di direzione. Questo parametro servirà successivamente per determinare la tortuosità minima
- Crea un nuovo `stateRoad` con le nuove coordinate, il numero di passi incrementato, la nuova direzione e il numero di cambi di direzione
- Aggiunge il nuovo percorso alla lista `roads` e la ritorna

findPercorso

La funzione `findPercorso` implementa un algoritmo BFS per trovare il percorso minimo tra due punti del piano, tenendo conto degli ostacoli. Restituisce `true` e il numero minimo di cambi di direzione se esiste un percorso, altrimenti `false` e -1.

Nel dettaglio:

La funzione è un metodo del tipo `piano` che prende in input le coordinate dei punti di partenza e di destinazione `(ax,ay)` e `(bx,by)`

Verifico che i punti passati come coordinate non siano dentro ad ostacoli, altrimenti interrompe subito l'esecuzione. Questo ottimizza ed evita ricerche inutili come cercare di spostare automi in un punto dentro ad un ostacolo

```
func (p piano) findPercorso(ax, ay, bx, by int) (bool, int) {
    if p.isOstacolo(ax, ay) || p.isOstacolo(bx, by) {
        return false, -1
    }
```

Viene calcolata la distanza Manhattan con `manhattanDist` e se essa è 0 (ovvero il punto di partenza e destinazione coincidono) ritorna positivamente, evitando anche qui ricerche inutili ed ottimizzando

```
d := manhattanDist(Punto{ax, ay}, Punto{bx, by})
if d == 0 {
    return true, 0
}
```

Vengono poi inizializzate: una coda `queue` contenente lo stato iniziale del percorso in formato `stateRoad`, una mappa `visited` che tiene traccia dei nodi che sono già stati visitati e il numero di cambi di direzioni avvenuti (tortuosità), ed infine un intero `minTurns` che tiene traccia del numero minimo di cambi di direzione quindi la tortuosità minima, partendo da -1 restituito se non c'è tortuosità

```
queue := []stateRoad{{ax, ay, 0, -1, 0}}
visited := make(map[visitedNodes]int)
minTurns := -1
```

Successivamente viene implementata la BFS vera e propria:

```
for len(queue) > 0 {
    current := queue[0]
    queue = queue[1:]
    if current.x == bx && current.y == by {
        if current.steps == d {
            if minTurns == -1 || current.turns < minTurns {
                minTurns = current.turns
            }
        }
        continue
    }
    if current.steps >= d {
        continue
    }
    roads := p.possibleRoads(current, bx, by, d)
    for _, road := range roads {
        key := visitedNodes{vX: road.x, vY: road.y, vDir: road.dir}
        if v, ok := visited[key]; ok && v <= road.turns {
            continue
```

```
        }
        visited[key] = road.turns
        queue = append(queue, road)
    }
}
```

La BFS opera nel seguente modo

- Un ciclo `for` continua finchè la coda non è vuota, estraendo il primo elemendo dalla coda ed assegnandolo alla variabile `current`
- Se il punto `current` corrisponde con le coordinate di destinazione e il numero di passi è corretto (ovvero il percorso ha la distanza minima `d` che abbiamo calcolato precedentemente con `manhattanDist`), aggiorna il numero minimo di cambi direzione definito come `minTurns` se il numero di `current.turns` è minore o se non è ancora stato assegnato
- Se il numero di passi supera la distanza `d`, passo al prossimo elemento. Così facendo evito di continuare a cercare strade se la distanza corretta è già stata trovata, ottimizzando
- Utilizza la funzione `possibleRoads` precedentemente descritta per calcolare tutti i possibili percorsi e salvarli all'interno di `roads`
- Itera grazie ad un `for` ogni percorso, e per ognuno di essi verifica se il nodo è già stato visitato con un numero di cambi di direzione minore od uguale. Se si, continua con il prossimo percorso
- Il percorso viene aggiunto a `queue` e la mappa `visited` dei nodi visitati viene aggiornata

Infine, se `minTurns` è ancora uguale a -1 significa che non è stato trovato alcun percorso valido, quindi la funzione restituisce `false` e -1, altrimenti restituisce `true` e il numero minimo di cambi di direzione `minTurns`

```
if minTurns == -1 {
    return false, -1
}
return true, minTurns
```

richiamo

La funzione `richiamo` sposta gli automi con un prefisso dato verso un punto specifico sul piano. Ecco una descrizione dettagliata di ogni sezione della funzione

Nel dettaglio:

La funzione è un metodo del tipo `piano` che prende in input le coordinate del punto di destinazione `(x,y)` e il prefisso del nome degli automi da richiamare `prefix`

```
func (p piano) richiamo(x, y int, prefix string) {
    var pq PriorityQueue
    for _, nome := range p.trie.searchPrefix(prefix) {
        a := p.automi[nome]
        d := manhattanDist(Punto{a.p.x, a.p.y}, Punto{x, y})
        if ok, _ := p.findPercorso(a.p.x, a.p.y, x, y); ok {
            heap.Push(&pq, DistanceRoad{name: nome, distance: d})
        }
    }
```

Dopo aver creato una variabile `pq` che rappresenta una coda a priorità, viene effettuata una ricerca degli automi tramite il prefisso `prefix` utilizzando la trienode `trie`

- Per ogni automa trovato, ottiene la sua posizione attuale `(a.p.x, a.p.y)`

- Calcola la distanza Manhattan con `manhattanDist` dal punto di arrivo `(x,y)`
- Verifica se esiste un percorso libero con `findPercorso` precedentemente descritta
- Se esiste un percorso, aggiunge l'automa nella coda a priorità `pq` con la distanza calcolata

Se la coda a priorità `pq` è vuota, la funziona termina evitando operazioni inutili

```
if pq.Len() == 0 {
    return
}
```

Infine gli automi selezionati vengono spostati

```
mDistance := pq[0].distance
for pq.Len() > 0 && pq[0].distance == mDistance {
    item := heap.Pop(&pq).(DistanceRoad)
    name := item.name
    if automa, ok := p.automi[name]; ok {
        point := Punto{automa.p.x, automa.p.y}
        if automi, ok := p.posAutomi[point]; ok {
            slice := removeElement(automi, name)
            if len(slice) == 0 {
                delete(p.posAutomi, point)
            } else {
                p.posAutomi[point] = slice
            }
        }
    }
    p.automi[name] = Automa{Punto{x, y}}
    p.posAutomi[Punto{x, y}] = append(p.posAutomi[Punto{x, y}], name)
}
```

- Viene ottenuta la distanza minima `mDistance` dalla coda con priorità `pq`
- Finchè ci sono automi nella coda con la stessa distanza minima il `for` estrae l'automa dalla coda
- Ottiene il nome dell'automa `name` e verifica se esiste nel piano
- Rimuove l'automa dalla posizione attuale del piano
- Aggiorna `piano` e `Trie` con la nuova posizione dell'automa

Scelte Scartate

Durante la modellazione del progetto e l'analisi dei problemi ho prodotto alcune soluzioni o idee che poi sono state scartate per diversi motivi ad esempio:

- Uso di una linked list per gli `Operatore`: non era più efficiente di una lista normale e appesantiva solo il programma
- Estrazione della posizione degli `Operatore` direttamente dalla lista senza usara la mappa di appoggio `occupati`: il programma diventava immensamente complesso quando poteva essere semplificato con una mappa di supporto
- Ricerca dei prefissi diretta senza `Trienode`: anche se funzionante, è molto più inefficiente rispetto all'uso di una Trienode per gestire i prefissi specialmente in situazioni con tanti automi da iterare

Costi Delle Operazioni

Costi Strutture dati

- Automi: Complessità spaziale O(1) per ogni istanza
- Ostacoli: Complessità spaziale O(1) per ogni istanza

- Occupati: Essendo una mappa per la ricerca di un elemento ha complessità temporale di O(1), e complessità spaziale O(n) per n numero di automi. Inserire o rimuovere un ostacolo in occupati ha costo temporale di O((x1 - x0 + 1) * (y1 - y0 + 1))
- Posizione Automi: Essendo una mappa la complessità è uguale ad Occupati
- Trienode:
  - Tempo:
    - insert: O(m), con m = lunghezza del nome (binario)
    - searchPrefix = O(m + k), con m = lunghezza del prefisso e k = numero di automi con quel prefisso
    - collectAll = O(n), con n = numero totale di automi
    - Totale: O(m + k + n)
  - Spazio:
    - Ogni nodo ha al massimo 2 figli (0 e 1) → O(n * m) per n automi con nomi lunghi m

Costi Algoritmi

- possibleRoads:
  - Complessità temporale e spaziale uguale a O(1) per il numero fisso di movimenti possibili
- findPercorso:
  - Tempo:
    - O(n) per il manhattanDistance per n automi
    - O(n) dove n è il numero di nodi visitati
  - Spazio:
    - O(n) per la coda e la mappa dei nodi visitati
- richiamo
  - Tempo:
    - O(n) per il manhattanDistance per n automi
    - O(k * (m+n)), dove k è il numero di automi con il dato prefisso, m è la lunghezza del prefisso, n è il numero di nodi visitati
    - O(d) per il BFS verso la posizione di richiamo
    - Totale: O(k * (m + n) + d)
  - Spazio: O(k + n) per la queue e la mappa di nodi visitati
- tortuosità
  - Tempo:
    - O(d) dove d è la distanza
  - Spazio:
    - O(d)

Test ed Esempi di Esecuzione

Per testare il progetto ho effettuato moltissimi test su tutti i vari aspetti presenti, per verificare la correttezza del codice e per metterlo alla prova

Alcuni dei test effettuati sono stati i seguenti:

- `c` (crea): creazione di un piano, inserimento ostacoli ed automi, verificare che si svuoti correttamente
- `S` (stampa): S di nulla, S di automi ed ostacoli
- `a` (creazione automa): a in punto vuoto, in un punto già occupato da ostacolo, con nome già esistente verificare che si sposti
- `o` (creazione ostacolo): o in punto vuoto, in punto già occupato da automa, verifica che gli ostacoli si soprappongano correttamente, che la mappa `occupati` rifletta la posizione degli ostacoli
- `s` (stato punto): crea un automa ed un ostacolo e verifica che le stampe di A O ed E siano corrette
- `r` (richiamo): su un punto ad automa libero, con ostacoli ad occupare automa, con un prefisso uguale al nome dell'automa, con un automa contornato da ostacoli, con un prefisso che non corrisponde a nessun automa, r con automi tutti bloccati, r su automa non esistente

- `p` (posizioni): p su un prefisso di nessun automa, su un prefisso di un singolo automa, di più automi
- `e` (esiste percorso): e su un percorso di partenza ed arrivo uguali, su un percorso valido senza ostacoli, su un percorso con ostacoli, su un automa non esistente
- `t` (tortuosità): t da punto a punto uguale, tra punti con percorso rettilineo, tra due punti con una curva, tra due punti con più curve, tra due punti bloccati da ostacolo, su automa non esistente

Questi test hanno dati tutti risultati positivo. Ho anche realizzato uno script `generator.go` per generare in maniera automatica dei test nel formato richiesto in modo da poter velocemente generare input di grandi dimensioni nel formato esatto. Alcuni esempi di test sono i seguenti:

Test 1

Il primo caso è un esempio molto semplice, ovvero un automa circondato completamente da ostacoli, nella cartella chiamato `29407A_corrado_francesco_test1IN`

```
c
a 5 3 1
o 0 0 10 2
o 0 4 10 6
o 0 0 4 8
o 6 0 9 7
S
r 16 3 1
S
f
```

L'output ottenuto è quello previsto:

```
(
1: 5,3
)
[
(0,0)(10,2)
(0,4)(10,6)
(0,0)(4,8)
(6,0)(9,7)
]
(
1: 5,3
)
[
(0,0)(10,2)
(0,4)(10,6)
(0,0)(4,8)
(6,0)(9,7)
]
```

Ovvero l'automa in posizione (5,3) anche dopo il richiamo non può raggiungerlo, in quanto circondato da ostacoli

Test 2

Il secondo caso invece è stato generato automaticamente con lo script realizzato, anche in questo caso riporto un caso molto semplice, nella cartella chiamato `29407A_corrado_francesco_test2IN`

```
c
t 3 84 0101
t -41 53 1010
t 83 4 1000
r 17 -68 0101
r -84 -6 0101
S
o 53 -66 -100 14
r 9 65 1110
t -44 -45 0000
o 57 50 -58 -83
o -48 -80 23 96
o -24 38 66 51
o -7 11 83 36
S
a 35 27 111000111
o 49 76 -58 -15
o -61 -40 85 -64
S
r 98 -100 0100
r 50 -43 1001
a 50 -28 1010011011
S
a -93 29 1110100
o 78 64 -29 27
t -91 37 0011
r -89 4 1010
S
a 67 -23 1110000111
a -92 -75 101111100
t 17 -60 0010
S
r 3 -91 0101
t -67 -14 0110
t -16 -19 0100
a -63 85 10100111
a 97 31 10001111
a 87 70 1001001110
o -32 80 65 -36
S
a 98 37 101000100
o 97 -64 -5 44
t 21 -39 1101
a -93 -55 101001000
o -29 75 -77 -4
S
t 9 -99 1111
o 88 -79 -42 58
r -18 4 1111
o -93 -16 65 -56
a 88 67 1110
r -87 -77 1000
```

```
t -61 75 0011
S
r -45 8 1010
t 12 -24 1100
r 24 43 1001
o 54 28 52 -60
r -76 60 0100
S
f
```

I file di test generati come si può vedere sono formati da blocchi di istruzioni varie nel formato richiesto. L'output del seguente test, che non riporto in quanto troppo lungo ma presente nella cartella come `29407A_corrado_francesco_test2OUT`

Test 3 e 4

Nella relativa cartella sono anche presenti due test nominati `29407A_corrado_francesco_test3IN` e `29407A_corrado_francesco_test4IN`. Questi test sono stati generati per mettere alla prova il tempo di testing e la capacià del programma su grandi input.

Il test `29407A_corrado_francesco_test3IN` genera da 600 righe di input 9350 righe di output in circa 0,3 - 0,4 sec

L'ultimo test `29407A_corrado_francesco_test4IN` invece è molto impegnativo con un input di 5994 righe, generando un output di circa 778.000 righe in in circa 70 secondi senza interruzioni su particolari punti nel programma ne problemi, verificandone l'efficienza

Con tutti i test qui descritti sono stato quindi in grado di testare sia la correttezza che l'efficienza del programma