Note Aggiuntive
(Relazione Progetto Algoritmi)

Kevin Manca, matricola 978578

Revisione 05 Agosto 2024

Contents

1 Introduzione

Note relative alla relazione e al progetto "Piastrelle Digitali", contenente le modifiche e revisioni fatte sul codice sorgente per risolvere problemi e discrepanze emerse durante la discussione in data 25 Luglio 2024.
Insieme a queste note, e alla relazione aggiornata, ho allegato un file contenente i "*diff*" rispetto al codice sorgente consegnato in data 10 Luglio 2024.

2 Funzioni modificate

In questa sezione sono presenti solo le funzioni che sono state modificate, con i cambiamenti effettuati ed eventuali argomentazioni relative ad essi.

2.1 spegni

Nella funzione `spegni` originale non era presente un controllo del *booleano* che comportava un crash del programma se questo veniva lanciato, passando ad esempio un input contenente una piastrella non esistente.

Versione originale

```
func spegni(p piano, x int, y int) {
 P := punto{x, y}
 if tile, ok := p.piastrelle[P]; tile.intensità > 0 {
```

```
  tile.intensità = 0
 }
}
```

Versione originale

```
func spegni(p piano, x int, y int) {
 P := punto{x, y}
 if tile, ok := p.piastrelle[P]; ok && tile.intensità > 0 {
  tile.intensità = 0
 }
}
```

2.2 propaga

La funzione *propaga* sfrutta la funzione calcolaColoriIntorno per memorizzare i colori dell'intorno prima di applicare qualsiasi *regola di propagazione* nella mappa *coloriIntorno*.
Viene passata poi questa mappa alla funzione verificaRegola che, come indicato nell'apposita sezione, restituisce la regola in esame se può essere soddisfatta dai colori dell'intorno, altrimenti restituirà `nil`.
Se una regola è stata restituita, quindi è valida, verifico che la piastrella nel *vertice* da cui effettuare la propagazione esista già e che sia accesa (intensità > 0), ed in quel caso ottengo la sua intensità per utilizzarla come valore per la funzione *colora()*, altrimenti sarà impostata ad 1 come predefinita (come da specifica).
Questa verifica è risultata necessaria in quanto non facendola, come in precedenza, l'intensità veniva impostata sul valore di una piastrella che magari esisteva già ma era stata spenta, il che portava a propagare il colore su più piastrelle ma con intensità a 0 (prendendo come intensità quella della piastrella spenta).

Versione originale

```
func propaga(p piano, x int, y int) {
 vertice := punto{x, y}

 intensità := 1

 for _, regolaDaValidare := range p.regole {
  if regolaValida := verificaRegola(p, regolaDaValidare, vertice, &intensità); regolaValida !=
   colora(p, x, y, regolaValida.colore, intensità)
   regolaValida.consumo++
   break
  }
 }
}
```

Versione modificata

```
func propaga(p piano, x int, y int) {
vertice := punto{x, y}

coloriIntorno := calcolaColoriIntorno(p, vertice)

for _, regolaDaValidare := range p.regole {
 if regolaValida := verificaRegola(regolaDaValidare, coloriIntorno); regolaValida != nil {
  intensità := 1
  if piastrella, esiste := p.piastrelle[vertice]; esiste && piastrella.intensità > 0 {
```

```
    intensità = piastrella.intensità
   }
   colora(p, x, y, regolaValida.colore, intensità)
   regolaValida.consumo++
   break
  }
 }
}
```

2.3 propagaBlocco

Come la funzione *propaga*, anche la funzione `propagaBlocco` sfrutta la funzione calcolaColoriIntorno per memorizzare i colori dell'intorno prima di applicare qualsiasi *regola di propagazione* nella mappa *coloriIntorno*.
Nella versione originale di questa funzione erano presenti alcuni errori, nel dettaglio: non era presente un *break* per interrompere l'applicazione della prima regola trovata per ciascuna *piastrella* appartenente al *blocco*; non utilizzo la funzione `colora()` in quanto la *propagaBlocco* cambia eventualmente solo il colore delle piastrelle.
Per questo motivo, utilizzo una mappa `coloriBloccoDaModificare` (*map[punto]string*) dove memorizzo le coordinate della piastrella da modificare con la stringa del nuovo colore, sulla quale itero successivamente, assegnando il nuovo colore a quello della piastrella presente nel relativo vertice.
Non è presente, inoltre, il controllo sulla lunghezza in quanto risulta superfluo visto l'utilizzo del *for-range*.

Versione originale

```
func propagaBlocco(p piano, x int, y int) {
 vertice := punto{x, y}

 _, ok := p.piastrelle[vertice]
 if !ok {
  return
 }

 visite := make(map[punto]bool)
 blocco := make(map[punto]*Piastrella)
 dfs(p, vertice, visite, blocco, false, nil)

 if len(blocco) > 0 {
  for vertice, piastrella := range blocco {
   if piastrella == nil {
    break
   }

   coordinate := punto{vertice.x, vertice.y}
   intensità := 1
   for _, regolaDaValidare := range p.regole {
    if regolaValida := verificaRegola(p, regolaDaValidare, coordinate, &intensità); regolaValid
     colora(p, coordinate.x, coordinate.y, regolaValida.colore, p.piastrelle[vertice].intensità
     regolaValida.consumo++
    }
   }
  }
```

```
 }
}
```

Versione modificata

```
func propagaBlocco(p piano, x int, y int) {
 vertice := punto{x, y}

 _, ok := p.piastrelle[vertice]
 if !ok {
  return
 }

 visite := make(map[punto]bool)
 blocco := make(map[punto]*Piastrella)
 dfs(p, vertice, visite, blocco, false, nil)

 coloriBloccoDaModificare := make(map[punto]string)

 for vertice := range blocco {
  coordinate := punto{vertice.x, vertice.y}
  coloriIntorno := calcolaColoriIntorno(p, coordinate)

  for _, regolaDaVerificare := range p.regole {
   if regolaValida := verificaRegola(regolaDaVerificare, coloriIntorno); regolaValida != nil {
    coloriBloccoDaModificare[coordinate] = regolaValida.colore
    regolaValida.consumo++
    break
   }
  }
 }

 for vertice, nuovoColore := range coloriBloccoDaModificare {
  if piastrella, ok := p.piastrelle[vertice]; ok {
   piastrella.colore = nuovoColore
  }
 }
}
```

2.4 dfs

Nella funzione `dfs` ho spostato all'inizio il controllo della mappa *blocco* (che se presente, implica che stia usando la funzione per ottenere le piastrelle del blocco), in modo da aggiungere anche la piastrella iniziale, mentre prima non veniva aggiunta, portando in certi casi ad un calcolo errato delle piastrelle appartenenti al blocco.

Versione originale

```
func dfs(
 p piano,
 vertice punto,
 visite map[punto]bool,
 blocco map[punto]*Piastrella,
 omogeneo bool,
 sum *int,
```

```
) {
 visite[vertice] = true
 colore := p.piastrelle[vertice].colore

 for _, direzione := range direzioni {
  nuovoVertice := calcolaDeltaVertice(vertice, direzione.x, direzione.y)

  if circonvicina, ok := p.piastrelle[nuovoVertice]; ok && !visite[nuovoVertice] && circonvicin
   if !omogeneo || (omogeneo && circonvicina.colore == colore) {
    if blocco != nil {
     blocco[vertice] = circonvicina
    }
    if sum != nil {
     *sum += circonvicina.intensità
    }
    dfs(p, nuovoVertice, visite, blocco, omogeneo, sum)
   }
  }
 }
}
```

Versione modificata

```
func dfs(
 p piano,
 vertice punto,
 visite map[punto]bool,
 blocco map[punto]*Piastrella,
 omogeneo bool,
 sum *int,
) {
 visite[vertice] = true
 colore := p.piastrelle[vertice].colore

 if blocco != nil {
  blocco[vertice] = p.piastrelle[vertice]
 }

 for _, direzione := range direzioni {
  nuovoVertice := calcolaDeltaVertice(vertice, direzione.x, direzione.y)

  if circonvicina, ok := p.piastrelle[nuovoVertice]; ok && !visite[nuovoVertice] && circonvicin
   if !omogeneo || (omogeneo && circonvicina.colore == colore) {
    if sum != nil {
     *sum += circonvicina.intensità
    }
    dfs(p, nuovoVertice, visite, blocco, omogeneo, sum)
   }
  }
 }
}
```

2.5 piastrelleCirconvicine

La funzione `piastrelleCirconvicine` è stata semplificata, ed ora calcola e popola una mappa (*map[punto]*Piastrella*) che poi restituisce, contenente appunto tutte le coordinate e i puntatori alle piastrelle circonvicine a quella presente nel vertice, passato come argomento alla funzione e preso come vertice di origine (o partenza).
È stato rimosso il calcolo dei colori dell'intorno e la relativa mappa *colori*, lasciando l'onere di quel calcolo ad una funzione ad-hoc chiamata `calcolaColoriIntorno`, e questo ha permesso anche di rimuovere un argomento alla funzione.

Versione originale

```
func piastrelleCirconvicine(p piano, vertice punto, colori map[string]int) (vicine map[punto]*P
 vicine = make(map[punto]*Piastrella)

 for _, direzione := range direzioni {
  nuovoVertice := calcolaDeltaVertice(vertice, direzione.x, direzione.y)

  if piastrella, ok := p.piastrelle[nuovoVertice]; ok {
   vicine[nuovoVertice] = piastrella
   if colori != nil {
    colori[piastrella.colore]++
   }
  }
 }
 return vicine
}
```

Versione modificata

```
func piastrelleCirconvicine(p piano, vertice punto) (vicine map[punto]*Piastrella) {
 vicine = make(map[punto]*Piastrella)

 for _, direzione := range direzioni {
  nuovoVertice := calcolaDeltaVertice(vertice, direzione.x, direzione.y)

  if piastrella, ok := p.piastrelle[nuovoVertice]; ok {
   vicine[nuovoVertice] = piastrella
  }
 }
 return vicine
}
```

2.6 verificaRegola

La funzione `verificaRegola` è stata semplificata ed è stato rimosso il calcolo delle piastrelle circonvicine.
In questa erano presenti alcuni errori, come quello di ricalcolare ad ogni iterazione le piastrelle circonvicine con la omonima funzione che popolava anche i colori, e ciò portava ad un calcolo errato per le funzioni `propaga` e `propagaBlocco`, in quanto l'intorno non veniva memorizzato nella versione originale, ma veniva ricalcolato ogni volta, riportando quindi eventuali modifiche di colori e/o altri comandi ricevuti che avevano alterato la situazione del *piano*.
Adesso la funzione riceve in input una mappa contenente i colori dell'intorno (*coloriIntorno*

map[string]int) che è stata memorizzata in precedenza e semplicemente verifica per la regola che deve essere valutata se i colori dell'intorno possono soddisfare tale regola.
In caso positivo restituisce la regola (un puntatore ad essa), altrimenti restituisce `nil`.

Versione originale

```
func verificaRegola(p piano, regola *Regola, vertice punto, intensità *int) *Regola {
 valoriColore := make(map[string]int)
 piastrelleCirconvicine(p, vertice, valoriColore)

 for colore, val := range regola.valColore {
  if valoriColore[colore] < val {
   return nil
  }
 }

 piastrella, piastrellaOk := p.piastrelle[vertice]
 if piastrellaOk {
  *intensità = piastrella.intensità
 }

 return regola
}
```

Versione modificata

```
func verificaRegola(regola *Regola, coloriIntorno map[string]int) *Regola {
 for colore, val := range regola.valColore {
  if coloriIntorno[colore] < val {
   return nil
  }
 }
 return regola
}
```

2.7 calcolaPistaBreve

In questa funzione, lo scopo è calcolare la pista più breve (se esiste), partendo da una piastrella di origine (*verticeOrig*) ed una di destinazione (*verticeDest*), e restituirne la *lunghezza*.
Erano presenti alcuni errori in questa funzione che portavano ad un calcolo errato della lunghezza della pista, a seconda dell'input.
La prima modifica è stata quella di fermare direttamente la computazione, restituendo 0 se almeno uno (o entrambi) i vertici passati come argomento della funzione non corrispondevano a piastrelle esistenti o accese ($intensità > 0$).
La *lunghezza* la calcolo ora come mappa da *punto* a intero, associandola quindi alla distanza da un certo vertice (quello visitato in precedenza).
Per calcolare correttamente la distanza, è necessario memorizzare quella dal vertice precedente e sommarla a 1 (nel caso del progetto parliamo di piastrelle con lato unitario) quando arrivo ad un nuovo vertice.
Esempio

```
Pista breve A - B - C
lunghezza[A] = 1  <- vertice di origine esistente, imposto a 1 la distanza
lunghezza[B] = lunghezza[A] + 1
lunghezza[C] = lunghezza[B] + 1 <- è il vertice di destinazione, calcolo e restituisco la dista
return lunghezza[C]
```

Imposto quindi il valore della mappa sulle coordinate della piastrella attuale come somma tra il valore della distanza del vertice precedente (*lunghezza[vertice]*) e 1.

Ho aggiornato poi la chiamata alla funzione `piastrelleCirconvicine` con la signature della funzione aggiornata come indicato nella sezione piastrelleCirconvicine, che richiede come argomenti solo il *piano* e il *vertice* da cui calcolare le piastrelle circonvicine.
Ho inoltre spostato il controllo del vertice che, se corrisponde a quello di destinazione, ferma la computazione e restituisce la lunghezza nell'iterazione sulle piastrelle adiacenti, il che rende la funzione leggermente più efficiente, non dovendo fare l'operazione di *pop* sulla coda (*queue*) (estrarre il primo elemento e rimuoverlo dalla coda, tramite subslicing) se sono arrivato nel vertice di destinazione.
Infine, se le coordinate della piastrella attuale (*coordinatePiastrella*) corrispondono a quelle della piastrella di destinazione (*verticeDest*), mi fermo e restituisco la lunghezza calcolata del vertice attuale (*lunghezza[coordinatePiastrella]*).

Versione originale

```
func calcolaPistaBreve(p piano, verticeOrig punto, verticeDest punto) (lunghezza int) {
 piastrellaOrig, origineOk := p.piastrelle[verticeOrig]
 piastrellaDest, destOk := p.piastrelle[verticeDest]

 if (!origineOk || piastrellaOrig.intensità == 0) || (!destOk || piastrellaDest.intensità == 0)
  lunghezza = 0
 } else {
  lunghezza = 1
 }

 visitate := make(map[punto]bool)
 queue := []punto{verticeOrig}

 visitate[verticeOrig] = true

 for len(queue) > 0 {
  vertice := queue[0]
  queue = queue[1:]

  if vertice == verticeDest {
   return lunghezza
  }

  adiacenti := piastrelleCirconvicine(p, vertice, nil)
  for _, piastrella := range adiacenti {
   coordinatePiastrella := punto{piastrella.x, piastrella.y}
   if !visitate[coordinatePiastrella] {
    queue = append(queue, coordinatePiastrella)
    visitate[coordinatePiastrella] = true
   }
  }
  lunghezza++
 }
 return 0
}
```

Versione modificata

```
func calcolaPistaBreve(p piano, verticeOrig punto, verticeDest punto) int {
 piastrellaOrig, origineOk := p.piastrelle[verticeOrig]
 piastrellaDest, destOk := p.piastrelle[verticeDest]

 if (!origineOk || piastrellaOrig.intensità == 0) || (!destOk || piastrellaDest.intensità == 0)
  return 0
 }

 visitate := make(map[punto]bool)
 queue := []punto{verticeOrig}
 lunghezza := make(map[punto]int)

 visitate[verticeOrig] = true
 lunghezza[verticeOrig] = 1

 for len(queue) > 0 {
  vertice := queue[0]
  queue = queue[1:]

  adiacenti := piastrelleCirconvicine(p, vertice)
  for _, piastrella := range adiacenti {
   coordinatePiastrella := punto{piastrella.x, piastrella.y}
   if !visitate[coordinatePiastrella] {
    queue = append(queue, coordinatePiastrella)
    visitate[coordinatePiastrella] = true
    lunghezza[coordinatePiastrella] = lunghezza[vertice] + 1

    if coordinatePiastrella == verticeDest {
     return lunghezza[coordinatePiastrella]
    }
   }

  }
 }
 return 0
}
```

2.8 calcolaColoriIntorno

Questa funzione non era presente inizialmente, ma l'ho creata per svolgere il compito di definire i colori dell'intorno e restituire la mappa *string* a *int*, dove la stringa rappresenta l'etichetta del colore e l'intero corrisponde al valore di quel colore nell'intorno.
Nelle funzioni chiamanti poi memorizzerò tale mappa che corrisponderà alla situazione dei colori del piano nella condizione originale, prima di effettuare comandi che alterano la situazione del piano.

Funzione

```
func calcolaColoriIntorno(p piano, vertice punto) (coloriIntorno map[string]int) {
 coloriIntorno = make(map[string]int)
 for _, piastrella := range piastrelleCirconvicine(p, vertice) {
  coloriIntorno[piastrella.colore]++
 }
```

```
 return coloriIntorno
}
```

3 Altro

È stata rimossa anche una struct *elemRegola* che non veniva utilizzata e non era stata rimossa in precedenza.
Sono stati aggiunti inoltre dei *file di test* nella relativa cartella.