Relazione di Algoritmi e Strutture Dati

Matteo Ghidini, Nicolas Cavicchioli, Francesco Lozio
Università degli Studi di Brescia, anno 2025

Indice

1. Introduzione

La presente relazione descrive l'implementazione del progetto per il corso di **Algoritmi e Strutture Dati** relativo all'anno accademico 2024/2025. L'obiettivo principale del progetto è stato lo sviluppo e la valutazione di un sistema per la navigazione su griglia. Questo problema di ricerca del cammino, ampiamente studiato in settori come la robotica, la logistica e la pianificazione di percorsi, richiede soluzioni efficienti e robuste, in particolar modo in presenza di ostacoli. Il lavoro si è concentrato sull'implementazione di un algoritmo specifico per la gestione di griglie complesse e per il calcolo dei percorsi ottimali.

Il progetto si è articolato in quattro fasi principali, ciascuna affrontando un aspetto critico della problematica:

- **Generazione dell'ambiente**: Creazione di una griglia di lavoro con l'inserimento di ostacoli di diverse tipologie, simulando un ambiente reale in cui operare.
- **Analisi topologica della griglia**: Calcolo del Complemento e del Contesto di un'origine, elementi fondamentali per comprendere la connettività e le proprietà spaziali dell'ambiente.
- **Implementazione dell'algoritmo di ricerca del cammino**: sviluppo di un algoritmo, basato sullo pseudo-codice fornito, per calcolare il percorso più breve da un punto d'Origine ad una Destinazione.
- **Sperimentazione e valutazione delle prestazioni**: Esecuzione di sperimentazioni sistematiche per misurare l'efficienza e la correttezza del progetto, utilizzando le griglie generate nella prima fase.

Nelle sezioni successive la relazione presenterà in dettaglio il lavoro svolto per ciascuno dei compiti elencati. La Sezione 2 descrive la creazione delle Griglie, la Sezione 3 l'implementazione del calcolo di Complemento e Contesto, la Sezione 4 l'algoritmo di ricerca del Cammino e la Sezione 5 i risultati sperimentali e l'analisi delle prestazioni.

A. Strumenti Utilizzati

L'elaborato è stato realizzato mediante l'uso di Java versione 21, avvalendosi di una libreria pre-esistente per la lettura di file JSON. La struttura del file JSON viene poi descritta nella sezione 2.1.

B. Stato cella

Per ogni cella della Griglia serve memorizzare il suo stato, che racchiude informazioni quali se la Cella è navigabile o meno, se appartiene al contesto o al complemento, o se è una cella di frontiera. Alcuni stati sono mutualmente esclusivi (un ostacolo non può appartenere al contesto), mentre altri ammettono delle sovrapposizioni (le celle di Landmark sono un sottoinsieme delle celle di Frontiera). Perciò si può rappresentare tale stato con un intero, associando i singoli bit a uno stato particolare secondo la seguente tabella:

Nome	value	mask
CONTESTO	00000001	00000001
REGINA	00000011	00000010
ORIGINE	00000111	00000100
COMPLEMENTO	00001000	00001000
FRONTIERA	00010000	00010000
LANDMARK	00110000	00100000
DESTINAZIONE	01000000	01000000
OSTACOLO	10000000	10000000
CHIUSURA	00000000	00001001

Tabella I: Valori binari per rappresentare lo stato di una cella

Dove i bit presenti nella colonna $value$ vengono usati per assegnare uno o più stati a una cella, mentre quelli nella colonna $mask$ vengono usati per controllare l'appartenenza di un intero a un particolare stato. Ad esempio, lo stato `CHIUSURA` non ha bit assegnati al campo $value$ perché non è possibile assegnare lo stato `CHIUSURA` a una cella, ma è possibile verificare se una cella appartenga ad essa controllando la presenza del bit associato al contesto o al complemento. In questo modo, controllare che una cella sia navigabile, che appartenga alla chiusura o che sia una cella di frontiera, lo si può esprimere in un'unica espressione:

$$(stato \,\&\, mask) > 0 \tag{1}$$

In altre parole, si controlla che almeno un bit specificato in $mask$ sia presente nello stato da controllare. Gli stati di ogni cella vengono memorizzate in un intero anziché in un byte perché la JVM gestisce la memoria con un allineamento a 4 byte [1], quindi utilizzare dei dati primitivi quali `byte` o `short` piuttosto che `int` non comporta un effettivo beneficio.

2. Compito Uno

A. Personalizzazione della Griglia

Per consentire la personalizzazione dei parametri, la griglia viene creata mediante un file JSON descritto in Figura 15.

B. Tipologie di Ostacoli

Ci sono 8 tipologie di Ostacoli possibili ed ognuno di essi ha un valore binario associato:

Nome	Valore binario	Descrizione
Semplice	000000001	Una singola cella non attraversabile
Agglomerato	000000010	Forma rettangolare con riempimento
Barra Verticale	000000100	Una retta verticale larga 1 ed altezza variabile
Barra Orizzontale	000001000	Una retta orizzontale alta 1 e larghezza variabile
Barra Diagonale	000010000	Una retta diagonale di misura variabile
Zona Chiusa	000100000	Forma rettangolare senza riempimento
Delimitatore Verticale	001000000	Una retta verticale larga 1 ed altezza massima
Delimitatore Orizzontale	010000000	Una retta verticale alta 1 e larghezza massima
Personalizzato	100000000	Tipologia personalizzata

Tabella II

Le coordinate della posizione della prima Cella di ciascun ostacolo vengono scelte casualmente grazie alla funzione `Random` di Java che, dato in ingresso un intero, genera una sequenza pseudo-casuale da cui estrarre valori interi. L'uso di questo intero per la generazione della sequenza pseudo-casuale consente di riprodurre la stessa Griglia grazie allo stesso valore del suddetto intero.
Il numero di Ostacoli può essere impostato attraverso l'apposito file JSON, tenendo conto che il numero inserito è il numero massimo di Ostacoli: nel caso in cui la generazione pseudo-casuale generi un Ostacolo su una o più celle non attraversabili, il generatore di Ostacoli esegue un ulteriore tentativo per trovare una nuova posizione pseudo-casuale con nuove dimensioni (qualora sia prevista la generazione di una dimensione dal valore pseudo-casuale).
Questo processo continua per un numero limitato di volte, al fine di evitare una rigenerazione illimitata dei valori dell'Ostacolo.
La tipologia Ostacolo Personalizzato viene usata per determinare il tipo delle disposizioni di Ostacoli specifiche realizzate per motivi di Sperimentazioni.

C. Tipo di una Griglia

Il tipo di una Griglia è dettato dal massimo numero di Ostacoli diversi da 0: tutti i tipi di Ostacoli che soddisfano questa condizione sono fattori di un `OR` che detta il tipo della Griglia. Ad esempio, se solo Ostacolo Semplice e Delimitatore Orizzontale hanno un numero massimo diverso da 0, allora il tipo Griglia è:

$$00000001 \vee 10000000 = 10000001. \quad (2)$$

Nel caso in cui si desideri invece creare una Griglia prefabbricata, non potendo sapere a priori quali Ostacoli siano stati usati per la creazione, viene assegnato un valore personalizzato che è 999. Il tipo delle disposizioni specifiche supera il 255 per evitare conflitti con i tipi già esistenti. Nello specifico, i tipi usati sono:

- `Variazione Dimensioni`: 256;
- `Spirale`: 257;
- `Linea Spezzata`: 258;
- `Doppia Linea Spezzata`: 259;
- `Scacchiera`: 260.

D. Pseudo-Codice

Algorithm 1 CreaGriglia(path)

```
▷ Se "path" non è un cammino valido, viene restituita una griglia di larghezza pari a 50, altezza pari a 50 e priva di Ostacoli
if not isPercorsoValido(path) then
    return creaGrigliaVuota(50, 50)
end if
▷ Conversione da path a file JSON che contiene i dati della Griglia
json ← letturaFile(path)
▷ Ottenimento dei dati necessari per la realizzazione della Griglia desiderata
w ← ottieniLarghezzaGriglia(json)
h ← ottieniAltezzaGriglia(json)
▷ Date le dimensioni, è ora possibile creare una Griglia con solo Celle attraversabili
griglia ← creaGrigliaVuota(w, h)
ostacoli ← ottieniNumeroMassimoOstacoli(json)
seed ← seed[json]
▷ Dato il numero massimo per ciascuna tipologia di Ostacoli, vengono inseriti gli Ostacoli nella Griglia
if isTipoSpecifico(ostacoli) then
    griglia ←costruzioneSpecifica(griglia)
else
    griglia ←generazioneStandard(griglia, ostacoli, seed)
end if
return griglia
```

Il metodo `generaCelle` è implementato da *tipo* ed è specifico per ogni tipo di Ostacolo. Di seguito, viene riportato

Algorithm 2 Generazione degli Ostacoli

```
1: function GENERAZIONESTANDARD(griglia, ostacoli, seed)
2:     ▷ il tipo Ostacolo PERSONALIZZATO è ignorato
3:     for tipo ∈ ostacoliEnum do
4:         ▷ il metodo numeroValidoDiOstacoli controlla che il numero di Ostacoli nella variabile ostacoli per il dato tipo sia maggiore di 0
5:         if NUMEROVALIDODIOSTACOLI(ostacoli,tipo) then
6:             celle ← GENERACELLE(tipo, griglia, seed)
7:             ADDOBSTACLE(griglia, celle, value[tipo])
8:         end if
9:     end for
10:    return griglia
11: end function
```

il metodo `generaCelle` specifico per Ostacolo Semplice come esempio.

Algorithm 3 GeneraCelle(griglia, seed)

```
1: finito ← false
2: retry ← 0
3: MAX_RETRY ← 15
4: celle ← ∅
5: while not finito do
6:     if retry ≥ MAX_RETRY then
7:         finito ← true
8:     else
9:         ▷ il metodo getCoordinataCasuale genera un numero intero positivo casuale entro il limite fornito
10:        x ← GETCOORDINATACASUALE(width[griglia])
11:        y ← GETCOORDINATACASUALE(height[griglia])
12:        if ISNAVIGABILE(griglia[y][x]) then
13:            nuovaCella ← (valore[OSTACOLO], x, y)
14:            celle ← celle ∪ nuovaCella
15:            finito ← true
16:        else
17:            retry ← retry + 1
18:        end if
19:    end if
20: end while
21: return celle
```

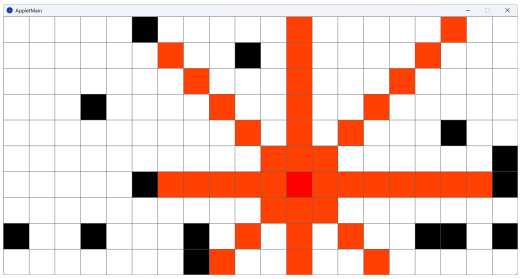
Algorithm 4 CreaGrigliaConOrigine(griglia, O)

```
1: in ← new int[height[griglia]][width[griglia]]
2: res ← new int[height[griglia]][width[griglia]]

3: for i ∈ [0, height[griglia]] do
4:     for j ∈ [0, width[griglia]] do
5:         in[i][j] ← stato[griglia[i][j]]
6:         res[i][j] ← 0
7:         if in[i][j] IS OSTACOLO then
8:             res[i][j] ← OSTACOLO
9:         end if
10:    end for
11: end for

12: chiusura ← ∅
13: PAINT(res, in, chiusura, x[O], y[O], REGINA, 0, -1, true,
    COMPLEMENTO, -1,-1, 1,-1)
14: PAINT(res, in, chiusura, x[O], y[O], REGINA, 0, 1, true,
    COMPLEMENTO, -1, 1, 1, 1)
15: PAINT(res, in, chiusura, x[O], y[O], REGINA, 1, 0, true,
    COMPLEMENTO, 1,-1, 1, 1)
16: PAINT(res, in, chiusura, x[O], y[O], REGINA, -1, 0, true,
    COMPLEMENTO, -1,-1, -1, 1)
17: PAINT(res, in, chiusura, x[O], y[O], REGINA, 1, 1, true,
    CONTESTO, 1, 0, 0, 1)
18: PAINT(res, in, chiusura, x[O], y[O], REGINA, 1, -1, true,
    CONTESTO, 1, 0, 0,-1)
19: PAINT(res, in, chiusura, x[O], y[O], REGINA, -1, -1, true,
    CONTESTO, -1, 0, 0,-1)
20: PAINT(res, in, chiusura, x[O], y[O], REGINA, -1, 1, true,
    CONTESTO, -1, 0, 0, 1)
    res[y[O]][x[O]] = ORIGINE
21: chiusura ← chiusura ∪ {(ORIGINE, x, y)}
22: frontiera ← CREAFRONTIERA(res)

23: return res, O, chiusura, frontiera, tipo[griglia]
```

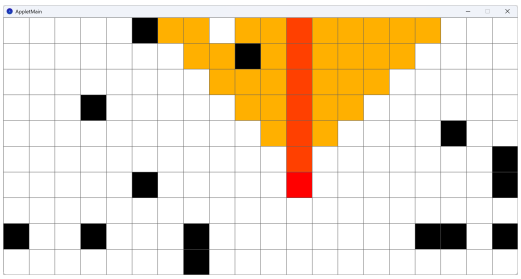
Partendo dalla Cella di Origine si esplorano tutte le direzioni cardinali e diagonali, assegnando a tali Celle lo stato `REGINA`[1] come in figura 1a. Ognuna delle Celle con stato `REGINA` viene usato come punto di partenza per una seconda esplorazione con direzione allineata[2], assegnando a queste Celle lo stato `COMPLEMENTO` se l'esplorazione secondaria usa mosse diagonali, mentre `CONTESTO` viene usato con direzioni cardinali (come nelle figure 1b e 1c). Vengono effettuate prima tutte le esplorazioni scrivendo il complemento, e successivamente tutte quelle del contesto, in questo modo se una cella è raggiungibile con un cammino libero di entrambi i tipi essa apparterrà al contesto (Figura 1d).

[1]Il nome deriva dai possibili movimenti dell'omonima pedina del celebre gioco degli scacchi, vista la verosimiglianza con essa

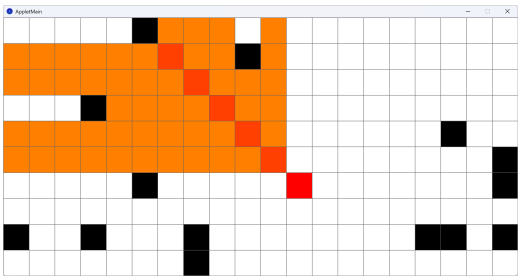
[2]Per direzione allineata si intende che l'angolo tra le due direzioni è di 45°



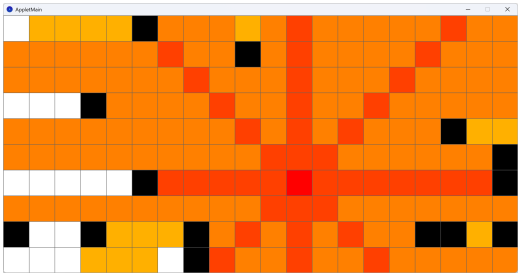
(a) celle con stato `REGINA`



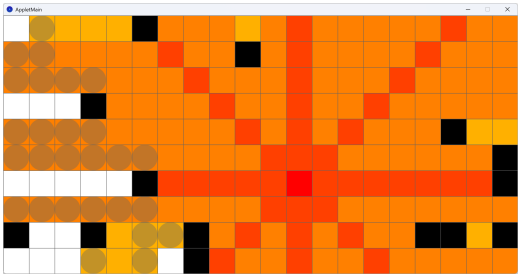
(b) celle con stato `COMPLEMENTO`



(c) celle con stato `CONTESTO`



(d) chiusura



(e) chiusura con celle di Frontiera

Figura 1

Infine, per ogni Cella della Chiusura si controlla se essa sia anche di Frontiera verificando se esiste una Cella ad essa adiacente tramite una mossa cardinale o diagonale che sia navigabile ma esclusa dalla Chiusura (Figura 1e).

Algorithm 5 Paint(out, in, chiusura, x, y, col, dx, dy, recursion, col2, dx1, dy1, dx2, dy2)

```
1:  while true do
2:      x ← x+dx
3:      y ← y+dy
4:      if x<0 or y<0 or x≥width[in] or y≥height[in] or stato[in[y][x]] IS OSTACOLO then
5:          return
6:      end if
7:      out[y][x] ← col
8:      chiusura ← chiusura ∪{(col, x, y)}
9:      if recursion then
10:         PAINT(out, in, chiusura, x, y, col2, dx1, dy1, false, 0, 0, 0, 0, 0)
11:         PAINT(out, in, chiusura, x, y, col2, dx2, dy2, false, 0, 0, 0, 0, 0)
12:     end if
13: end while
```

Algorithm 6 CreaFrontiera(res, griglia)

```
1:  frontiera ← ∅
2:  for (i, j) ∈ [0, height[griglia]] × [0, width[griglia]] do
3:      for (i2, j2) ∈ [i − 1, i + 1] × [j − 1, j + 1] do
4:          if i2 < 0 or j2 < 0 or i2 ≥ height[griglia] or
5:  j2 ≥ width[griglia] then
6:              break
7:          end if
8:          if res[i2][j2] IS not CHIUSURA and res[i2][j2] IS OSTACOLO then
9:              res[i][j] = FRONTIERA
10:             frontiera ← frontiera ∪ {(j, i)}
11:         end if
12:     end for
13: end for
14: return frontiera
```

A. Distanza libera

La distanza di un Cammino Libero, sia che esso sia di tipo uno o di tipo due, è formata da un numero intero non negativo di passi cardinali e da un numero intero non negativo di passi diagonali, definite rispettivamente d_{torre} e $d_{alfiere}$. Il vantaggio di memorizzare i due interi in aggiunta alla distanza è di ridurre il più possibile i calcoli e i confronti da fare con rappresentazioni in floating-point, permettendo così una maggiore accuratezza nei valori ottenuti.

In assenza di ostacoli è possibile calcolare in tempi costanti i valori che d_{torre} e $d_{alfiere}$ assumono con la seguente formula:

$$\begin{aligned} distanza(O, D) &= d_{torre}(O, D) + \sqrt{2} \cdot d_{alfiere}(O, D) \\ d_{alfiere}(O, D) &= \min\{\Delta_x, \Delta_y\} \\ d_{torre}(O, D) &= \max\{\Delta_x, \Delta_y\} - \min\{\Delta_x, \Delta_y\} \\ \Delta_x = |O.x - D.x| \qquad & \Delta_y = |O.y - D.y| \end{aligned} \tag{3}$$

B. Regioni di celle navigabili

Si può osservare come, a seguito della creazione della Chiusura a partire da una Cella d'Origine, le Celle navigabili non appartenenti alla Chiusura solitamente creino delle regioni compatte di Celle ancora da esplorare (Figura 2), e una sola di esse conterrà la Cella di Destinazione. Perciò se si riuscisse ad individuarla sarebbe possibile limitare l'analisi delle Celle di Frontiera a quelle adiacenti alla suddetta regione.

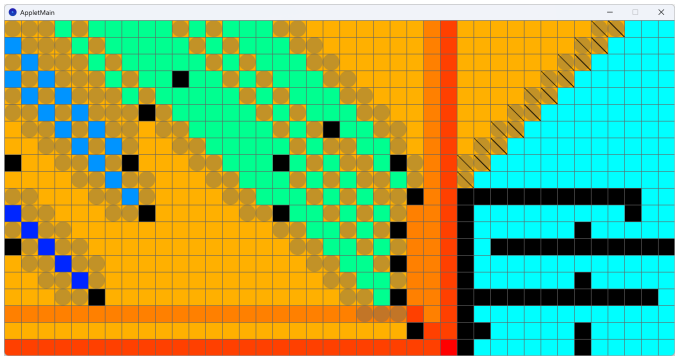


Figura 2: Colorazione delle regioni di celle da esplorare, con evidenziate le celle di frontiera adiacenti alla regione in azzurro

Come si potrà notare nello pseudocodice 11 del compito tre la chiamata ricorsiva richiede di creare una nuova griglia. Questo permette, una volta individuata la regione di interesse, di utilizzare una sotto-griglia contenente tale regione di dimensioni non maggiori di quella di partenza, consentendo di ridurre il numero di Celle di Frontiera da considerare.

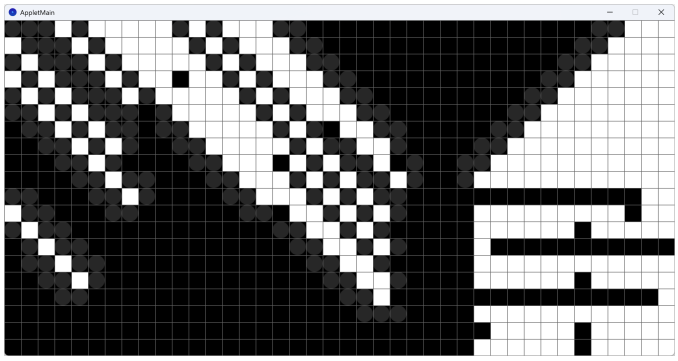


Figura 3: Esempio di Griglia alla seconda iterazione ricorsiva di cammoniMin, dove la Chiusura di partenza è diventata un'ostacolo

Questo approccio richiede però l'utilizzo di una **struttura dati alternativa**, dove l'obbiettivo è sempre quello di ottenere l'intero che rappresenta lo stato di una Cella date le sue coordinate bidimensionali intere, ma questo deve essere possibile per qualunque sotto-griglia diversa da quella di partenza, mentre l'implementazione tramite matrice ha implicitamente il vincolo di dover contenere la cella di posizione $(0, 0)$. Per questo motivo viene introdotta la `BiHashmap<K,V>`, ossia un dizionario che mappa una prima chiave a un secondo

dizionario, il quale a sua volta mappa una seconda chiave al valore da associare alle due chiavi.

Algorithm 7 RegioneContenente(griglia, x, y)

```
indexer ← CREABIHASHMAP()
for each (x, y) ∈ griglia do
    indexer[y][x] ← stato[griglia[y][x]]
    if stato[griglia[y][x]] IS OSTACOLO then
        indexer[y][x] ← -1
    end if
end for
regione = CREAREGIONE(griglia[y][x]);
BUCKETPAINT(griglia, indexer, 1, regione, x, y)
return regione
```

Questo algoritmo invoca la funzione ricorsiva `bucketPaint`, la quale ha un funzionamento analogo all'algoritmo 5 e ricorda l'algoritmo di esplorazione in profondità di un grafo [2]: Partendo da una cella iniziale si analizzano di volta in volta le celle ad essa adiacente fin tanto che si trova in una posizione valida, sia una cella navigabile non appartenente alla chiusura e che non abbia una regione ad essa assegnata.

Algorithm 8 BucketPaint(griglia, indexer, idx, reg, x, y)

```
if x<xmin[griglia] or y<ymin[griglia] or x>xmax[griglia]
or y>ymax[griglia] then
    return
end if
if GETORDEFAULT(indexer, y, x, -1)≥0 then
    return
end if
c ← griglia[y][x]
if stato[c]≠0 then
    if c IS FRONTIERA and c ∉ frontiera[reg] then
        frontiera[reg] ← frontiera[reg] ∪ c
    end if
    return
end if
indexer[y][x] ← idx
reg ← reg ∪ c
BUCKETPAINT(griglia, indexer, idx, reg, x−1, y−1)
BUCKETPAINT(griglia, indexer, idx, reg, x,   y−1)
BUCKETPAINT(griglia, indexer, idx, reg, x+1, y−1)
BUCKETPAINT(griglia, indexer, idx, reg, x−1, y)
BUCKETPAINT(griglia, indexer, idx, reg, x,   y)
BUCKETPAINT(griglia, indexer, idx, reg, x+1, y)
BUCKETPAINT(griglia, indexer, idx, reg, x−1, y+1)
BUCKETPAINT(griglia, indexer, idx, reg, x,   y+1)
BUCKETPAINT(griglia, indexer, idx, reg, x+1, y+1)
```

Algorithm 9 CreaSottoGriglia(griglia, regione)

```
(width,height) ← (maxWidth[regione],maxHeight[regione])
(xmin,ymin) ← (xmin[regione],ymin[regione])
map ← ∅

for each (i, j) ∈ [0, height] × [0, width] do
    map[i][j] ← stato[griglia[i+ymin][j+xmin]]
end for

return map, width, height, xmin, ymin, tipo[griglia]
```

C. Precisazioni

1) L'utilizzo della struttura dati alternativa è necessaria per rappresentare sotto-griglie di qualunque tipo, forma e dimensione;
2) Essendo la ricerca di un elemento in una `BiHashmap` non costante in termini di tempo, un'implementazione di Compito Due con il **solo** doppio dizionario risulta peggiorativa sia dal punto di vista spaziale che temporale;
3) Questa implementazione si è rivelata controproducente per griglie di dimensioni elevate, dove le chiamate ricorsive, per la creazione della regione, causavano nella maggior parte dei casi `StackOverflowError`, per questo motivo per per griglie grandi verranno effettuate sperimentazioni con solo l'implementazione tramite matrice.

Algorithm 10 GrigliaFrontieraPair(compitoDueStrategy, griglia, O, D)

```
if name[compitoDueStrategy] = "V0" then
    chiusura ← CONVERTICHIUSURAINOSTACOLO(griglia)
    g ← ADDOBSTACLE(griglia, chiusura)
    return g, frontiera[g]
end if

regione ← REGIONECONTENENTE(griglia, x[D], y[D])
frontieraList ← FRONTIERA(regione, O)
g ← CREASOTTOGRIGLIA(griglia, regione)
return g, frontieraList
```

4. COMPITO TRE

Algorithm 11 CamminoMin(griglia, O, D, compito2)

```
g ← CREAGRIGLIACONORIGINE(griglia, O)
g2, frontiera ←GRIGLIAFRONTIERAPAIR(compito2, g, O, D)
if F ∈ contesto[g] then
    return DLIB(O, D), ⟨(O, 0), (D, 1)⟩
end if
if F ∈ complemento[g] then
    return DLIB(O, D), ⟨(O, 0), (D, 2)⟩
end if
if frontiera = ∅ then
    return ∞, ⟨⟩
end if
lunghezzaMin ← ∞
seqMin ← ⟨⟩
for each (F, t) ∈ frontiera do
    lf ← DLIB(O, F)
    if lf <DLIB(O, F) then
        lFD, seqFD ← CAMMINOMIN(g2, F, D)
        lTot ← lF + lFD
        if lTot < lunghezzaMin then
            lunghezzaMin ← lTot
            seqMin ← COMPATTA(⟨(O, 0), (F, t)⟩, seqFD)
        end if
    end if
    return lunghezzaMin, seqMin
end for
```

A. Introduzione al Compito Tre

Il Compito Tre si concentra sull'implementazione di un algoritmo per la determinazione del Cammino Minimo tra una Cella di Origine (O) e una Cella di Destinazione (D) all'interno di una Griglia bidimensionale. L'algoritmo gestisce Griglie rettangolari con e senza Ostacoli.

L'implementazione del Cammino Minimo è realizzata dalla classe `CompitoTreImplementation`, che espone il metodo principale `camminoMin(...)`. Questo metodo agisce da punto di ingresso per il calcolo, delegando la logica ricorsiva a `calcoloCamminoMin` e gestendo l'inizializzazione, le statistiche e le interruzioni. È possibile fornire come argomento un `ICompitoDue`, modificando la logica di calcolo e gestione della Griglia.

B. calcoloCamminoMin

Il metodo `calcoloCamminoMin` rappresenta il nucleo ricorsivo dell'algoritmo di ricerca del Cammino Minimo. La sua struttura riflette un approccio di esplorazione basato su un concetto di Frontiera.

C. Flusso di Esecuzione e passi fondamentali

La logica ricorsiva di `calcoloCamminoMin` segue i seguenti passaggi:

1) **Preparazione alla Ricorsione:** All'inizio di ogni chiamata ricorsiva, il livello di ricorsione viene incrementato e viene eseguito un controllo per eventuali richieste di interruzione (`gestoreInterruzioni.checkInterruzione()`).
2) **Creazione della Griglia con Origine:** Viene utilizzata l'implementazione del CompitoDue (`ICompitoDue compitoDue`) per generare una `IGrigliaConOrigine`. La creazione di questa Griglia computa anche il calcolo della Chiusura, assegnando il valore di Contesto o Complemento alle Celle.
3) **Caso Base (Destinazione Raggiunta):** L'algoritmo verifica se la Cella di Destinazione (D) è nel Contesto o nel Complemento rispetto all'origine corrente, che si traduce in una verifica dello stato Chiusura. Se la condizione `condizioneCasoBase(griglia, destinazione)` è soddisfatta, significa che la Destinazione è stata raggiunta o è direttamente raggiungibile partendo dall'Origine. Viene costruito e restituito un `Cammino` dalla O corrente alla D.
4) **Estrazione della Frontiera:** Se il Caso Base non è soddisfatto, l'algoritmo identifica la Frontiera della Griglia.
5) **Caso Frontiera Vuota (Destinazione Irraggiungibile):** Se la frontiera è vuota, significa che non ci sono Celle da esplorare dalla posizione corrente per raggiungere la Destinazione. In questo scenario, il percorso è considerato irraggiungibile, e viene restituito un Cammino con lunghezza infinita e una sequenza di Landmark vuota.
6) **Esplorazione Ricorsiva della Frontiera:** Per ogni cella F nella frontiera, l'algoritmo procede ricorsivamente:

   - **Creazione della Griglia**: la corrente Griglia viene copiata e modificata, viene aggiunta la chiusura della stessa come ostacolo per impedire cicli. Questa griglia *(g2)* viene utilizzata per i successivi calcoli.
   - **Calcolo della Frontiera**: viene calcolata la Frontiera.
   - **Condizione di Ottimizzazione**: viene applicata una condizione (presente a riga 16/17 dello pseudocodice fornito) per decidere se proseguire la ricorsione su F. Questa condizione serve a potare il ramo di ricerca nel caso in cui è garantito che il cammino parziale non possa raggiungere una lunghezza inferiore a quella più piccola trovata fino a quel punto.
   - **Chiamata Ricorsiva**: se la condizione è soddisfatta, viene effettuata una chiamata ricorsiva a *calcoloCamminoMin(g2, F, dest, stats, compitoDue)*. Il risultato di questa chiamata *(camminoFD)* rappresenta il Cammino Minimo dalla Cella F alla Destinazione D nella nuova Griglia con Ostacoli aggiornati.
   - **Aggiornamento del Cammino Minimo Globale**: La lunghezza del cammino parziale *(lF)* viene sommata alla lunghezza del Cammino trovato ricorsivamente *(camminoFD)*. Se la lunghezza totale *(lTot)* è minore della *lunghezzaMin* corrente, la

lunghezzaMin e la *seqMin* (la sequenza di Landmark che compone il Cammino) vengono aggiornati. Nel caso di due Cammini con medesima lunghezza, viene mantenuto quello con il numero di Landmark minore.

- **Gestione della Condizione non Soddisfatta**: Se la condizione non è soddisfatta, viene incrementato un contatore *(stats.incrementaIterazioniCondizione())*, indicando quante volte un percorso è stato potato preventivamente.

7) **Restituzione del Risultato:** Alla fine del ciclo su tutte le Celle della Frontiera, l'algoritmo restituisce il `risultatoFinale` (il Cammino più breve trovato) e il livello di ricorsione viene decrementato.
8) **Fine esecuzione**: il calcolo si conclude se nel livello 1 di ricorsione si esauriscono le Celle di Frontiera esplorabili. Viene restituito il `risultatoFinale` che è il Cammino Minimo tra Origine e Destinazione.

D. Aggiunte Funzionali e Ottimizzazioni

L'implementazione del Compito Tre incorpora diverse **aggiunte** che migliorano l'efficienza dell'algoritmo, gestite tramite un sistema di strategie, attivabili grazie a un sistema di configurazione (`CamminoConfiguration`).

1) Ordinamento della Frontiera:

- **Descrizione:** L'algoritmo offre la possibilità di ordinare le Celle di Frontiera prima di iterare su di esse. Questa funzionalità è gestita dalla classe `FrontieraStrategy`. Nel codice le Frontiere vengono ordinate per distanza non decrescente dalla Destinazione (`Utils.distanzaLiberaTra(F, dest)`).
- **Impatto:** L'ordinamento della frontiera può avere un impatto significativo sulle prestazioni. Dare priorità all'esplorazione delle celle più vicine alla destinazione (secondo un'euristica) può portare a trovare il Cammino Minimo più rapidamente, riducendo il numero totale di Celle visitate.
- **Dipendenze dagli ostacoli:** Il beneficio dell'ordinamento dipende fortemente dalla disposizione degli Ostacoli. In Griglie aperte, l'euristica può guidare efficacemente la ricerca. In griglie molto complesse l'euristica potrebbe non essere altrettanto efficace o addirittura fuorviante in alcuni casi, portando a esplorare percorsi apparentemente brevi che si rivelano peggiori. Dalle sperimentazioni si è notato che l'ordinamento della frontiera con Griglie complesse porta in media a rallentamenti invece che miglioramenti.

2) Condizione rafforzata della Frontiera:

- **Descrizione:** L'algoritmo può utilizzare una "condizione rafforzata" per filtrare le Celle della Frontiera da esplorare. Questa condizione è gestita dalla classe `CondizioneStrategy`. Viene aggiunto un limite inferiore di distanza al confronto: se `condizioneRafforzata` è abilitata, la condizione è `(lF + Utils.distanzaLiberaTra(F, dest) < lunghezzaMin)` invece che solo `(lF < lunghezzaMin)`
- **Impatto:** Questa condizione agisce come una potatura `(pruning)` della ricerca. Calcolando una stima della distanza rimanente dalla Cella F alla Destinazione `(Utils.distanzaLiberaTra(F, dest))` e sommandola alla distanza già percorsa `(lF)`, l'algoritmo può escludere rami di ricerca che, anche nel migliore dei casi (Cammino Libero dalla Frontiera alla Destinazione), supererebbero la lunghezza minima già trovata. Ciò riduce il numero di chiamate ricorsive e di Celle analizzate, accelerando il processo. Dalle sperimentazioni, in `Griglia a Spirale`, osserviamo che la modalità $PERFORMANCE\ CONDIZIONE\ RAFFORZATA$ ha ottenuto risultati ottimi in confronto alle altre modalità.

3) Cache dei cammini:

- **Descrizione:** L'implementazione include una cache basata su *HashMap* `(pathCache)` che memorizza i Cammini già calcolati. La cache, gestita dalla classe `CacheStrategy` può essere abilitata o disabilitata tramite la `CamminoConfiguration`. La chiave della cache è generata includendo non solo Origine e Destinazione, ma anche un hash che rappresenta lo stato attuale della Griglia (ovvero, la posizione degli Ostacoli). Il metodo `calcolaHashOstacoli` è responsabile di generare l'hash della griglia.

Algorithm 12 PutCammino

Require: Griglia g, origine o, destinazione d, cammino c
1: *chiave* $\leftarrow$ GeneraChiaveCache(g, o, d)
2: salva c in cache con *chiave*

Algorithm 13 GeneraChiaveCache

Require: Griglia g, cella origine o, cella destinazione d
Ensure: Chiave univoca per la cache
1: **return** coord(o) + "$\rightarrow$" + coord(d)
+"—" + CalcolaHashOstacoli(g)

Algorithm 14 CalcolaHashOstacoli

Require: Griglia g
Ensure: Valore hash basato sugli ostacoli
1: *listaOstacoli* $\leftarrow$ lista vuota
2: **for all** cella$[y][x]$ in g **do**
3: **if** *cella*$[y][x]$ IS `OSTACOLO` **then**
4: aggiungi $cella[y][x]$ a *listaOstacoli*
5: **end if**
6: **end for**
7: *hash* $\leftarrow$ hash(*listaOstacoli*)
8: **return** *hash*

- **Funzionamento:**
  1) Generazione chiave: La chiave $O.x(), O.y() -> D.x(), D.y() | hash\_ostacoli$ assicura che un Cammino in cache sia valido solo per la specifica configurazione di Griglia per cui è stato calcolato.
  2) Cache HIT: Se una ricerca per una data chiave viene richiesta e la chiave è presente in cache, il Cammino viene recuperato istantaneamente.
     Prima dell'inizio delle ricorsioni, l'algoritmo tenta di recuperare un risultato dalla cache. Se un Cammino per quella specifica configurazione è già stato calcolato e memorizzato, viene immediatamente restituito, prevenendo il ricalcolo del cammino parziale e migliorando significativamente le prestazioni.
  3) Cache MISS: Se la chiave non è presente, l'algoritmo procede con il calcolo, e il risultato finale (anche se rappresenta un cammino irraggiungibile) viene inserito nella cache per future richieste.
- **Benefici:** La cache è una delle ottimizzazioni più potenti per ridurre i tempi di calcolo in scenari dove le stesse sotto problematiche (Cammini tra coppie di punti su griglie identiche o con identica configurazione di Ostacoli) vengono risolte ripetutamente. Nelle sperimentazioni (Figura 14) si nota come le modalità che attivano la cache sono in media più rapide, in particolar modo in presenza di un elevato tasso di Celle di Frontiera per Landmark.

4) Svuota Frontiera:

- **Descrizione:** La strategia SvuotaFrontiera introduce un'ottimizzazione che mira a potare, o a "svuotare", la frontiera in base a una condizione di Cammino ottimale già noto. Questa strategia si attiva solo se è già stato trovato un Cammino Minimo. Quando la strategia è abilitata, l'algoritmo confronta la lunghezza del Cammino corrente con la lunghezza del Cammino Minimo già trovato (ottenuto grazie ai *monitor*). Se il Cammino attuale, pur essendo ancora in costruzione, ha già superato in lunghezza il Cammino Minimo conosciuto, l'algoritmo conclude che non è più possibile trovare una soluzione migliore a partire da quel percorso. Di conseguenza, la frontiera viene svuotata, interrompendo l'esplorazione di quel ramo di ricerca. Il codice gestisce due scenari principali:
  - Confronto diretto: Se la lunghezza totale del cammino in costruzione supera la lunghezza del Cammino Minimo già trovato, la frontiera viene svuotata.
  - Confronto parziale: Il codice considera anche il caso in cui il punto di partenza della ricerca attuale (O) si trovi su un Cammino già calcolato. In questo caso, viene ricalcolata la distanza dal punto di partenza originale fino a O (parte del cammino minimo già trovato) e confrontata con la lunghezza del cammino in costruzione. Nel caso in cui il cammino parziale si riveli peggiore a priori, la frontiera viene svuotata.
- **Impatto:** Svuotando la frontiera, l'algoritmo evita di esplorare percorsi che sono matematicamente subottimali. Questo riduce il numero di Celle analizzate e le chiamate ricorsive, accelerando la ricerca del Cammino più breve. È un'ulteriore forma di "potatura" (*pruning*) che si aggiunge alla *Condizione Rafforzata* della Frontiera. Nelle sperimentazioni, nella Griglia *Variazione Ostacoli* si può notare che la modalità $PERFORMANCE\ SVUOTA\ FRONTIERA$ riduce i tempi di esecuzione di un fattore pari a 3,45 rispetto alla modalità $DEFAULT$.

E. Riassunto e Gestione dell'Interruzione

1) Riassunto e Statistiche: L'algoritmo è dotato di un sistema di raccolta statistiche, implementato attraverso l'interfaccia `IStatisticheEsecuzione` e la classe `StatisticheEsecuzione`. Queste statistiche possono essere utilizzate per creare un riassunto grazie alla classe `RiassuntoFactory`.

- **Raccolta Dati Iniziale**: All'inizio dell'esecuzione del metodo `camminoMin`, vengono salvati i dati fondamentali relativi alla griglia: dimensioni (`height`, `width`), tipo (`getTipo`), e le coordinate di Origine e Destinazione.
- **Metriche Specifiche del Compito Tre**: Il riassunto finale include metriche chiave per valutare l'esecuzione:
  - **Celle Frontiera Visitate**: Contatore che registra quante celle della Frontiera sono state effettivamente analizzate e considerate per la ricorsione.
  - **Iterazioni Condizione**: Contatore che registra il numero totale di volte in cui la condizione di ottimizzazione (la "condizione alla riga 16/17") ha assunto il valore "falso", indicando quante esplorazioni sono state potate. Nella modalità di esecuzione del compito Tre si può leggere quale delle due condizioni (base o rafforzata) era attiva durante il calcolo.
  - **Stato Interruzione**: Indica se il calcolo è stato interrotto prematuramente (a causa di tempo scaduto o di una richiesta esplicita).
- **Statistiche delle Aggiunte**: Il riassunto incorpora anche lo stato delle "aggiunte" abilitate: se la cache era attiva, se l'ordinamento e potatura della frontiera sono stati abilitati.
- **Generazione del Report**: Il metodo `generaRiassunto(..)` in `StatisticheEsecuzione` compila tutte queste informazioni in una classe `Riassunto` (che contiene Tipo di riassunto e contenuto come `String`). Questo fornisce un feedback dettagliato sulle prestazioni e sul comportamento dell'algoritmo per ogni esecuzione. Il tipo di riassunto cambia in base alle necessità, da una modalità più verbosa a un file in formato CSV. Le diverse modalità sono contenute nell'`Enum TipiRiassunto` e possono essere istanziate tramite la `RiassuntoFactory`, indicando semplicemente il tipo richiesto.

2) Gestione dell'Interruzione: La classe `CompitoTreImplementation` implementa l'interfaccia `IInterrompibile`, fornendo due meccanismi per interrompere l'esecuzione del calcolo del Cammino Minimo in modo controllato. Questa funzionalità si dimostra vantaggiosa specialmente in contesti nei quali l'esecuzione potrebbe richiedere tempi di esecuzione elevati. La gestione è centralizzata nella classe `GestioneInterruzioni`.

1) **Interruzione su Timeout (`setTimeout`)**:
   - **Meccanismo**: Il metodo `setTimeout(long duration, TimeUnit unit))` permette di impostare un tempo massimo di esecuzione.
   - **Funzionamento**: Periodicamente, all'interno del loop di esplorazione della frontiera e all'inizio di ogni chiamata ricorsiva, viene invocato `checkInterruzione()`. Se il tempo trascorso supera il `timeoutMillis` specificato, viene sollevata una `InterruptedException`.
   - **Benefici**: Previene che l'algoritmo si blocchi indefinitamente su Griglie complesse, garantendo una risposta entro un limite di tempo prestabilito.
2) **Interruzione su Richiesta Esplicita (`interrupt`)**:
   - **Meccanismo**: Il metodo `interrupt()` attiva una variabile interna (`interrompiSuRichiesta = true`).
   - **Funzionamento**: Analogamente al timeout, il `checkInterruzione()` verifica anche lo stato di questa variabile. Se il valore assunto è `true`, viene sollevata una `InterruptedException`. Questo permette a un'entità esterna (es. un thread UI, un altro componente del sistema) di fermare il calcolo in qualsiasi momento.
   - **Benefici**: Offre un controllo sull'esecuzione, consentendo all'utente o al sistema di fermare un calcolo non più necessario o di cui si vuole ottenere un risultato parziale.

- **Gestione dell'Eccezione**: Entrambi i meccanismi di interruzione lanciano una `InterruptedException`. Il blocco `try-catch` nel metodo `camminoMin` cattura questa eccezione che viene gestita da `gestisciInterruzione_GeneraCammino(e)` Questo metodo tenta di recuperare e restituire il cammino parziale migliore trovato fino al momento dell'interruzione, ritornando un Cammino *infinito* se nessun Cammino è stato trovato. Le statistiche vengono aggiornate per riflettere l'interruzione.

3) Monitoraggio del progresso: La classe `CompitoTreImplementation` implementa l'interfaccia `IHasProgressoMonitor`. L'implementazione fornisce due monitor di progresso (`monitor` e `monitorMin`) che tracciano l'evoluzione del Cammino trovato dall'algoritmo. Questi monitor sono fondamentali per la visualizzazione in tempo reale, per la gestione delle interruzioni e per capire l'andamento del calcolo, soprattutto quando l'algoritmo impiega tempo significativo.

- **monitor**: Traccia il Cammino corrente della ricorsione attiva. Ogni volta che l'algoritmo ricade nel caso base o esegue una chiamata ricorsiva aggiorna il cammino parziale attualmente in fase di esplorazione.
- **monitorMin**: Traccia il miglior Cammino Minimo corrente. Questo è particolarmente utile perché anche se l'algoritmo non ha ancora terminato l'esplorazione completa, `monitorMin` fornirà il Cammino più breve conosciuto in quel momento.

Entrambi i monitor registrano il Cammino come una lista di `Landmark`, rendendoli utili per la visualizzazione grafica del progresso.

F. Stack dei Landmark

I monitor si avvalgono di una *pila* ausiliario *(stackCammino)* che traccia l'evolversi dell'algoritmo. Si può pensare come una riproduzione dello Stack delle chiamate del compilatore. Quando un monitor si aggiorna, prende informazioni dalla pila e ricostruisce il percorso. Questo è necessario dato che l'algoritmo ricorsivo usa un meccanismo di ricostruzione e compattazione per la soluzione finale ma non è possibile creare il percorso completo se non si ritorna alla chiamata iniziale. Usando la pila è facilmente calcolabile anche la lunghezza.

G. Uso delle Strategie per la flessibilità

Il design dell'algoritmo `camminoMin` in `CompitoTreImplementation` fa ampio uso del Design Pattern Strategy. Questo pattern consente di definire una famiglia di algoritmi, incapsularli e renderli intercambiabili. Ciò significa che il comportamento dell'algoritmo può essere modificato dinamicamente tramite configurazione, senza alterare la struttura del codice che li utilizza.

1) Interfacce e Implementazioni concrete delle Strategie: Le strategie sono definite tramite interfacce Java, e per ogni funzionalità esistono due implementazioni concrete:

- **DebugStrategy**:
  - **Scopo**: Gestisce l'output di debug.
  - **Implementazioni**: `DebugAbilitato` (mostra a video messaggi di debug) e `DebugDisabilitato` (non mostra a video nulla, ottimizzato per le prestazioni).
- **CacheStrategy**:
  - **Scopo**: Gestisce la logica di memorizzazione e recupero dei cammini dalla cache.
  - **Implementazioni**: `CacheAttiva` (interagisce con la classe `CamminoCache` per memorizzare e recuperare risultati) e `CacheNull` (implementazione di un "null object pattern" che non fa nulla).
- **CondizioneStrategy**:
  - **Scopo**: Implementa la logica per la "condizione rafforzata" o la "condizione base" per la potatura della ricerca.
  - **Implementazioni**: `CondizioneRafforzata` (applica un'euristica più stringente per potare i rami

di ricerca) e `CondizioneNormale` (applica la condizione base).

- **FrontieraStrategy**:
  - **Scopo**: Definisce come viene estratta e, opzionalmente, ordinata la lista delle celle di frontiera.
  - **Implementazioni**: `FrontieraOrdinata` (ordina le celle di frontiera in base a un criterio di distanza euristico) e `FrontieraNormale` (restituisce la frontiera senza un ordinamento specifico).
- **SvuotaFrontieraStrategy**:
  - **Scopo**: Definisce un'operazione aggiuntiva di potatura della frontiera.
  - **Implementazioni**: `SvuotaFrontieraAbilitato` (applica una potatura avanzata, controllando con i monitor la lunghezza del percorso minimo individuato) e `SvuotaFrontieraDisabilitato` (restituisce la frontiera non modificata).

2) StrategyBundle: Per semplificare la gestione e il passaggio delle diverse strategie, viene utilizzata la classe `StrategyBundle`. Questa classe è un semplice contenitore che raggruppa un'istanza di ogni tipo di strategia (Debug, Cache, Condizione, Frontiera, SvuotaFrontiera). Invece di dover gestire singolarmente cinque diverse strategie, l'algoritmo lavora con un'unica istanza di `StrategyBundle`.

3) Gestione della configurazione tramite Bit Flags (ConfigurationFlag, ConfigurationMode, CamminoConfiguration): La chiave di volta che permette la selezione dinamica delle strategie è il sistema di configurazione basato su bit flags:

- **ConfigurationFlag (Enum)**: Definisce singoli "flag" come potenze di 2. Questo permette di combinare più funzionalità in un unico valore intero (bitwise OR). Le flag presenti sono:
  - `DEBUG`: utilizzata per stampare su console diversi valori utili per debug del codice
  - `MONITOR_ENABLED`: utilizzata per gestire attivare i `Monitor` dell'applicazione; tutte le modalità attivano i monitor, dato che non ci sono vantaggi a disattivarli.
  - `SORTED_FRONTIERA`: per utilizzare la `FrontieraOrdinata` nella FrontieraStrategy
  - `CONDIZIONE_RAFFORZATA`: per utilizzare la `CondizioneRafforzata` nella CondizioneStrategy
  - `CACHE_ENABLED`: per utilizzare la `CacheAttiva` nella CacheStrategy
  - `SVUOTA_FRONTIERA`: per utilizzare la `SvuotaFrontieraAbilitato` nella SvuotaFrontieraStrategy
- **ConfigurationMode (Enum)**: Definisce impostazioni di configurazione predefinite a cui sono associate una combinazione specifica di `ConfigurationFlag` abilitati.
- **CamminoConfiguration (Classe)**: è la classe che gestisce la modalità di esecuzione di `CompitoTreImplementation`. Usando un intero per salvare le `flag` attive, è possibile creare modalità personalizzate di esecuzione oppure usare quelle predefinite presenti in `ConfigurationMode`. Permette aggiunta e rimozione di `flag`. Se si crea manualmente una modalità di esecuzione che non corrisponde a nessuna di quelle presenti, viene dato il nome `CUSTOM`, se invece corrisponde a una modalità presente nei preset, viene utilizzato il nome ufficiale della modalità.

ConfigurationMode	ConfigurationFlag
DEFAULT	MONITOR_ENABLED
DEBUG	DEBUG MONITOR_ENABLED
PERFORMANCE	
	MONITOR_ENABLED SORTED_FRONTIERA CONDIZIONE_RAFFORZATA CACHE_ENABLED
PERFORMANCE_CACHE	
	MONITOR_ENABLED CACHE_ENABLED
PERFORMANCE_NO_CACHE	
	MONITOR_ENABLED SORTED_FRONTIERA CONDIZIONE_RAFFORZATA
PERFORMANCE_SORTED_FRONTIERA	
	MONITOR_ENABLED SORTED_FRONTIERA
PERFORMANCE_NO_SORTED_FRONTIERA	
	MONITOR_ENABLED CONDIZIONE_RAFFORZATA CACHE_ENABLED
PERFORMANCE_CONDIZIONE_RAFFORZATA	
	MONITOR_ENABLED CONDIZIONE_RAFFORZATA
PERFORMANCE_NO_CONDIZIONE_RAFFORZATA	
	MONITOR_ENABLED SORTED_FRONTIERA CACHE_ENABLED
PERFORMANCE_SVUOTA_FRONTIERA	
	MONITOR_ENABLED SVUOTA_FRONTIERA
PERFORMANCE_FULL	
	MONITOR_ENABLED SORTED_FRONTIERA CONDIZIONE_RAFFORZATA CACHE_ENABLED SVUOTA_FRONTIERA

Tabella III: `ConfigurationFlag` per ogni `ConfigurationMode`

4) StrategyFactory: La classe `StrategyFactory` è il componente che collega la configurazione (`ConfigurationMode`) alle implementazioni concrete delle strategie. Il suo ruolo è quello di creare l'appropriato `StrategyBundle` in base alle `flag` abilitate nel `ConfigurationMode` corrente.

Durante l'inizializzazione di `CompitoTreImplementation` o ogni volta che la configurazione viene modificata (tramite `setConfiguration()`, `with()`, `without()`), il metodo `initializeStrategies()` viene invocato. Questo metodo utilizza `StrategyFactory` per creare un nuovo `StrategyBundle` che riflette la nuova configurazione.

5) Utilizzo delle Strategie in CompitoTreImplementation: Una volta che il `StrategyBundle` è stato creato e assegnato alla variabile `strategies` in `CompitoTreImplementation`, l'algoritmo invoca le diverse funzionalità tramite questo contenitore, senza conoscere le implementazioni concrete sottostanti.

Ad esempio:

- Per il debug:
  `strategies.getDebugStrategy().println(...);`
- Per la cache:
  `strategies.getCacheStrategy().get(...);` oppure
  `strategies.getCacheStrategy().put(...);`
- Per la condizione di frontiera:
  `strategies.getCondizioneStrategy()`
  `.isSoddisfatta(...);`
- Per la frontiera:
  `strategies.getFrontieraStrategy()`
  `.getFrontiera(...);`
- Per il svuota frontiera:
  `strategies.getSvuotaFrontieraStrategy()`
  `.isFrontieraDaSvuotare(...);`

Questo rende il codice di `calcoloCamminoMin` pulito e focalizzato sulla logica principale, delegando i dettagli implementativi specifici (come "ordinare la frontiera" o "usare la cache") alle rispettive strategie.

6) Vantaggi:

- **Flessibilità**: È possibile cambiare il comportamento dell'algoritmo (es. abilitare/disabilitare cache, debug, ordinamento) con una semplice modifica della configurazione.
- **Manutenibilità**: Ogni funzionalità specifica (es. caching, debug) è incapsulata nella propria strategia, rendendo più facile la modifica o l'estensione senza impattare altre parti del codice.
- **Verifica correttezza**: Le singole strategie possono essere testate in isolamento, semplificando il processo di verifica.
- **Leggibilità**: Il codice principale di `calcoloCamminoMin` è più pulito e facile da comprendere, in quanto i dettagli implementativi sono astratti dietro le interfacce delle strategie.

5. Compito Quattro

Per le analisi del Compito Quattro viene usata la convenzione secondo cui una griglia ha una dimensione $n \times m$ dove n indica la larghezza della Griglia mentre m è la sua altezza. Le analisi sono state valutate per tutte le implementazioni del Compito Tre.

A. Analisi spaziale

La firma del metodo accetta in ingresso: una griglia, una cella d'origine ed una cella di destinazione. La complessità spaziale è quindi un $O(n \cdot m) + O(1) + O(1) = O(n \cdot m)$.
Il metodo fa anche uso di uno *stack*, per contenere il cammino corrente come Landmarks, lo spazio massimo occupato è uguale alla profondità l della ricorsione più profonda, quindi $O(l)$. All'inizio dell'algoritmo, viene creata una nuova griglia in cui viene impostata la nuova origine. Questa nuova griglia viene salvata; dato che le dimensioni della griglia rimangono immutate nel peggiore dei casi, questa variabile contribuisce alla complessità spaziale con $O(n \cdot m)$. Se l'algoritmo non cade nel caso base immediatamente, viene creata una nuova variabile che tiene in memoria le celle di Frontiera. Assumendo che nel caso peggiore le celle di Frontiera siano $n \times m$, allora questa variabile contribuisce alla complessità con $O(n \cdot m)$. Quando si trasforma la Chiusura precedente in Ostacolo, viene creata una nuova Griglia con le medesime dimensioni e quindi, la complessità spaziale aumenta di $O(n \cdot m)$. Quindi lo spazio occupato dalle chiamate ricorsive per le griglie è $O(l) \cdot O(n \cdot m)$. Infine, quando si trova un nuovo potenziale cammino minimo, viene creato un Cammino e confrontato con l'attuale miglior Cammino. Quindi ci sono 2 Cammini salvati nello stesso momento. Ogni Cammino contiene una lista di Landmark ed il caso peggiore è quello in cui il numero di Landmark è $n \times m$. Quindi, la complessità spaziale viene incrementata di $2 \cdot O(n \cdot m)$. Totale: $O(n \cdot m) + O(l) + O(n \cdot m \cdot l) + 2 \cdot O(n \cdot m) = O(n \cdot m \cdot l)$.

Viene ora eseguita un'analisi sul caso migliore: la Destinazione si trova nel Contesto o nel Complemento di Origine. In questo caso, lo *stack* contiene solamente l'Origine e la Destinazione. Viene ugualmente creata una nuova Griglia ($O(n \cdot m)$) e i parametri in ingresso restano invariati ($O(n \cdot m)$); il Cammino calcolato contiene una lista di Landmark di 2 soli elementi che sono Origine e Destinazione ($O(1)$) e la profondità massima l è pari a 1. La complessità spaziale nel caso migliore è dunque di $O(n \cdot m)$.

B. Analisi temporale

La creazione di una nuova Griglia viene eseguita grazie al metodo di CompitoDue `calcola`. Questo metodo usa un doppio ciclo considerando larghezza ed altezza della griglia. Quindi la complessità di `calcola` viene incrementata di $O(n \cdot m)$. L'implementazione di `calcola` influisce su questa considerazione:

- utilizzando $V0$, la rigenerazione delle Griglie non fa variare le dimensioni delle stesse, rendendo la complessità temporale pari a $\theta(n \cdot m)$;
- utilizzando $V1$, la rigenerazione delle Griglie è in grado di escludere diverse Celle portando ad una possibile riduzione delle dimensioni. Pertanto, la complessità temporale si attesta a $O(n \cdot m)$.

Per semplicità di analisi verrà preso come punto di riferimento l'implementazione $V1$ per il metodo `calcola`.

Dunque, l'algoritmo itera su tutti gli elementi della griglia: $O(n \cdot m)$. Infine, viene nuovamente eseguito un doppio ciclo `for` su un array di dimensioni pari a quelle della griglia, quindi $O(n \cdot m) \cdot O(1)$ con la considerazione che gli altri due cicli interni invece vengono eseguiti al più 9 volte, viste le condizioni (che si traduce in O(1)). In totale, ogni volta che verrà creata una Griglia, con creaV0, sarà necessario sommare alla complessità temporale totale $O(n \cdot m) + O(n \cdot m) + O(n \cdot m) \cdot O(1) = O(n \cdot m)$. Tornando a CompitoTreImplementation ed assumendo che le operazioni di `push` e `pop` siano $O(1)$, si ha la creazione di una nuova Griglia con il metodo di Compito Due, quindi viene aggiunto $O(n \cdot m)$. Quando invece viene creata la lista di Frontiere, è possibile che essa venga ordinata in ordine non decrescente e successivamente trasformata in lista. Considerando che nel caso peggiore il numero di Frontiere sono $n \times m$, l'operazione di trasformazione è $O(n \cdot m)$ mentre l'operazione di ordinamento, implementato da Java usando Dual-Pivot Quicksort [3], ha una complessità temporale di $O(n \cdot m \cdot \log(n \cdot m))$.

Nel caso peggiore, la complessità temporale dipende in misura significativa dal numero di Celle di Frontiera presenti, per le quali non è possibile determinare un limite superiore in funzione delle sole dimensioni della Griglia. Come emerge dalle sperimentazioni, esiste infatti una correlazione tra il tempo di esecuzione e il numero di Celle di Frontiera.

Nel caso migliore, la Destinazione si trova nel Contesto o nel Complemento dell'Origine e quindi l'esecuzione si interrompe con successo prima della creazione della lista di Frontiere. Tutte le operazioni fino a questo punto sono $\Omega(1)$ (`push`), ottenimento di una Cella a specifiche coordinate, controllo dello Stato di una Cella per appartenenza a Contesto o Complemento) eccezion fatta per la creazione della nuova Griglia con il metodo di Compito Due, la cui complessità rimane invariata essendo $\Omega(n \cdot m)$.
Dunque, la complessità temporale totale dell'algoritmo nel caso migliore è $\Omega(n \cdot m)$.

C. Sperimentazioni

La sperimentazione delle differenti implementazioni è stata svolta usando tutte le possibili combinazioni di $ConfigurationMode$ e di $CompitoDueImpl$. Sono state usate differenti griglie per verificare i differenti casi sull'algoritmo. Tutte le verifiche hanno confermato la correttezza del risultato: la lunghezza del cammino minimo è la stessa sia per il percorso Origine-Destinazione che per il percorso Destinazione-Origine benché i due cammini non fossero uno l'opposto dell'altro. L'analisi si concentra sul tempo speso per trovare una soluzione, senza contare il tempo di verifica della correttezza. Per una corretta valutazione dei tempi, ogni istanza di Griglia viene eseguita 10 volte e si considera il tempo medio. Questo non accade in due casi:

- L'istanza ha raggiunto un tempo preimpostato e fisso a 15 minuti dopo il quale l'esecuzione viene interrotta;
- Nel primo tentativo, l'algoritmo stabilisce entro lo scadere del tempo che non è possibile raggiungere la Destinazione.

Analizzando i risultati, le due implementazioni di CompitoDue, ossia $V0$ e $V1$, presentano differenze in numero di Celle di Frontiera considerate, numero di condizioni di riga 16 dell'algoritmo attivate e massima profondità. Le analisi saranno strutturate come segue:

- Analisi approfondita delle combinazioni di tutte le implementazioni di CompitoTre con $CompitoDueImpl.V0$;
- Elenco delle stesse combinazioni ma con $CompitoDueImpl.V1$ con particolare attenzione sulla differenza nei dati d'esecuzione;
- Confronto diretto tra l'implementazione $PERFORMANCE\ FULL$ delle due implementazioni di CompitoDue per evidenziare la differenza nei dati e sottolineare queste ripercussioni nei risultati di $V1$.

Nelle tabelle viene inserita la categoria "Landmark Intermedi" con cui s'intende il numero di Landmark del Cammino Minimo esclusa la Destinazione. La categoria "CDF per Landmark" è il rapporto tra numero di Celle di Frontiera e numero di Landmark Intermedi.

1) Variazione Dimensioni: Mediante la variazione sulle dimensioni, è possibile verificare l'impatto delle sole dimensioni sul calcolo del cammino minimo. A tale scopo, viene scelta una Griglia con un solo ostacolo Delimitatore Verticale, posto al centro della Griglia, meno una Cella che consente al cammino di passare e raggiungere la destinazione usando un solo Landmark.

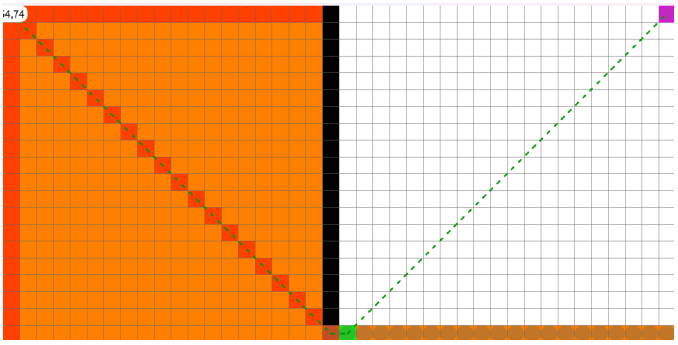


Figura 4

	Griglia		
Proprietà	piccola	media	grande
Dimensione ($w \times h$)	20x20	40x20	80x20
Spazio occupato (KB)	3,13	6,25	12,50
Celle di frontiera	1	21	43
Landmark Intermedi	1	1	1
Massima profondità	2	2	2
CDF per Landmark	1	21	43

Tabella IV

	piccola		media		grande	
Metriche	V0	V1	V0	V1	V0	V1
Tempo (ms)	0,117	0,204	1,211	2,011	2,881	3,662
Cache hit	0	0	0	0	0	0
Celle di Frontiera	1	1	21	21	43	43
Iterazioni Condizione	0	0	1	1	23	23
Spazio (KB)	3,13	3,13	6,25	6,25	12,50	12,50
Massima Profondità	2	2	2	2	2	2
CDF per Landmark	1	1	21	21	43	43

Tabella V

Analizzando i dati presenti nella tabella VI, è possibile stabilire che l'uso di $SORTED\ FRONTIERA$ si dimostra particolarmente efficace per la risoluzione di Cammini Minimi su Griglie di piccole dimensioni mentre negli altri casi è l'uso di $CONDIZIONE\ RAFFORZATA$ a fare la differenza dati gli eccellenti risultati con $PERFORMANCE\ CONDIZIONE\ RAFFORZATA$ e $PERFORMANCE\ NO\ CACHE$. La non rilevanza della cache è invece dovuta al Cammino Minimo scelto per questa valutazione: utilizzando un Cammino Minimo che utilizza un singolo Landmark, la cache non viene mai utilizzata, portando ad un'effettiva riduzione di prestazioni, dovuta alla scrittura del Cammino Parziale, senza un effettivo utilizzo.

Tenendo conto di variazioni dei tempi dovuti a differenti esecuzioni, il cambio da $V0$ a $V1$ non ha portato differenze statistiche: le migliori aggiunte al CompitoTre sono le stesse per $V0$ anche se è genericamente possibile affermare che per questa Griglia $V1$ è peggiore in termini di tempo a causa dell'aggiuntiva operazione di raggruppamento di Celle che non ha portato alcun beneficio a causa della struttura stessa della Griglia. In termini di spazio, non ci sono variazioni tra $V0$ e $V1$.

Viene ora preso in considerazione il confronto tra le varie Griglie con differenti implementazioni del CompitoDue usando l'implementazione $PERFORMANCE\ FULL$ di CompitoTre riportato nella Tabella V.

Come già evidenziato, non vi è alcuna variazione statistica per i due compiti ma l'operazione aggiuntiva di $V1$ ha portato peggioramenti. È dunque possibile concludere che optare per $CompitoDueImpl.V1$ non è conveniente quando si ha una Griglia relativamente semplice come quella presa in analisi.

Alla fine di tutte le considerazioni, è possibile confermare che la semplice variazione di dimensioni non è il fattore cruciale nella diminuzione delle prestazioni temporali: nonostante Griglia Grande abbia una larghezza 4 volte superiore a quella di Griglia Piccola, non è possibile notare l'incremento esponenziale atteso.

Ciò si nota anche per le prestazioni spaziali: essendo la stima dipende dalla Massima Profondità costante tra tutte le Griglie e dalle dimensioni delle stesse, il raddoppio di una sola di esse porta al raddoppio dello spazio occupato.

	piccola		media		grande	
Modalità tempi (ms)	V0	V1	V0	V1	V0	V1
DEFAULT	0,076	0,170	0,773	1,743	2,567	6,322
PERFORMANCE CACHE	0,067	0,209	1,294	2,082	4,616	7,178
PERFORMANCE CONDIZIONE RAFFORZATA	0,066	0,206	0,084	0,481	0,156	0,698
PERFORMANCE SORTED FRONTIERA	0,067	0,195	0,849	2,036	2,800	6,112
PERFORMANCE NO CACHE	0,051	0,186	0,767	1,698	1,511	2,969
PERFORMANCE NO CONDIZIONE RAFFORZATA	0,090	0,229	1,436	2,122	4,665	7,413
PERFORMANCE NO SORTED FRONTIERA	0,081	0,268	0,115	0,395	0,244	0,709
PERFORMANCE	0,108	0,209	1,490	2,107	2,233	3,882
PERFORMANCE SVUOTA FRONTIERA	0,092	0,178	0,936	1,955	2,362	6,389
PERFORMANCE FULL FRONTIERA	0,117	0,204	1,211	2,011	2,881	3,662

Tabella VI

2) Spirale: La Griglia Spirale è usata per verificare il comportamento dell'algoritmo con una Griglia raggiungendo un discreto numero di Landmark, al contrario di Variazione Dimensioni. Il comportamento atteso è quello di un aumento contenuto in termini di tempo d'esecuzione ma una crescita in termini di spazio occupato.

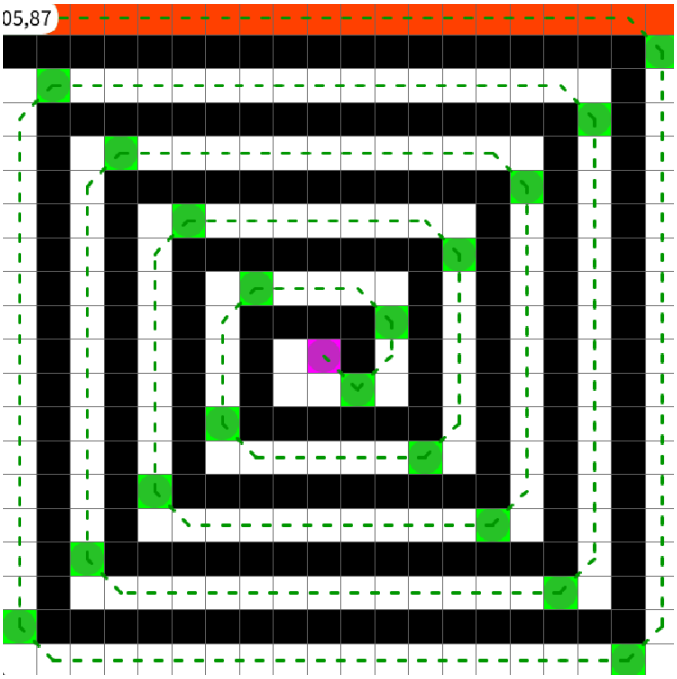


Figura 5

Le dimensioni utilizzate sono:

	Griglia		
Proprietà	piccola	media	grande
Dimensione ($w \times h$)	10x10	20x20	30x30
Spazio occupato (KB)	3,52	29,69	101,95
Celle di frontiera	8	18	28
Landmark Intermedi	7	17	27
Cache Hit	0	0	0
Iterazioni Condizione	0	0	0
Massima profondità	9	19	29
CDF per Landmark	1,142	1,058	1,037

Tabella VII

Vengono ora analizzate le differenti $ConfigurationMode$ usando $CompitoDueImpl.V0$:

Modalità-Dimensione	Tempo esecuzione (ms)
DEFAULT	
piccola	4,500
media	1,861
grande	3,497
PERFORMANCE CACHE	
piccola	0,163
media	0,701
grande	1,977
PERFORMANCE CONDIZIONE RAFFORZATA	
piccola	0,105
media	0,328
grande	0,838
PERFORMANCE SORTED FRONTIERA	
piccola	0,097
media	0,355
grande	0,923
PERFORMANCE NO CACHE	
piccola	1,227
media	0,591
grande	1,203
PERFORMANCE NO CONDIZIONE RAFFORZATA	
piccola	0,193
media	0,611
grande	2,330
PERFORMANCE NO SORTED FRONTIERA	
piccola	1,012
media	1,147
grande	2,130
PERFORMANCE	
piccola	0,159
media	0,619
grande	1,858
PERFORMANCE SVUOTA FRONTIERA	
piccola	0,144
media	0,352
grande	0,923
PERFORMANCE FULL	
piccola	0,158
media	0,779
grande	1,718

Tabella VIII

Spirale si è dimostrata una sperimentazioni molto semplice e ripetitiva: l'assenza di multiple Celle di Frontiera da considerare non richiede implementazioni aggiuntive per ottenere le migliori prestazioni ed è possibile evincere ciò sia dai singoli tempi d'esecuzione che dalle statistiche delle singole istanze che non differiscono tra di loro nonostante le aggiunte che dovrebbero facilitare certi comportamenti dell'algoritmo.

Si considerano ora i dati con $CompitoDueImpl.V1$:

modalità-dimensione	Tempo esecuzione (ms)	celle di frontiera	Dimensioni (KB)	Massima Profondità
DEFAULT				
piccola	0,136	8	3,52	9
media	0,705	18	29,69	19
grande	1,706	28	101,95	2
PERFORMANCE CACHE				
piccola	0,173	8	3,52	9
media	0,925	0	29,69	19.0
grande	2,349	28	101,95	29.0
PERFORMANCE CONDIZIONE RAFFORZATA				
piccola	0,147	8	3,52	9
media	0,698	18	29,69	19
grande	1,762	28	101,95	29
PERFORMANCE SORTED FRONTIERA				
piccola	0,139	8	3,52	9
media	0,661	18	29,69	19
grande	1,887	28	101,95	29
PERFORMANCE NO CACHE				
piccola	0,155	8	3,52	9
media	0,715	18	29,69	19
grande	1,877	28	101,95	29
PERFORMANCE NO CONDIZIONE RAFFORZATA				
piccola	0,210	8	3,52	9
media	0,842	18	29,69	19
grande	2,311	28	101,95	29
PERFORMANCE NO SORTED FRONTIERA				
piccola	0,167	0	3,52	9
media	0,829	18	29,69	19
grande	2,277	28	101,95	29
PERFORMANCE				
piccola	0,188	8	3,52	9
media	0,859	18	29,69	19
grande	2,265	28	101,95	29
PERFORMANCE SVUOTA FRONTIERA				
piccola	0,151	8	3,52	9
media	0,697	18	29,69	19
grande	1,917	28	101,95	29
PERFORMANCE FULL				
piccola	0,192	8	3,52	9
media	0,912	18	29,69	19
grande	2,288	28	101,95	29

Tabella IX

Viene ora effettuato un confronto tra le varie Griglie con differenti implementazioni del CompitoDue usando l'implementazione $PERFORMANCE\ FULL$ di CompitoTre:

	piccola		media		grande	
Metriche	V0	V1	V0	V1	V0	V1
Tempo (ms)	0,158	0,192	0,779	0,912	1,718	2,288
Cache hit	0	0	0	0	0	0
Celle di Frontiera	8	8	18	18	28	28
Iterazioni Condizione	0	0	0	0	0	0
Spazio (KB)	3,52	3,52	29,69	29,69	101,95	101,95
Massima Profondità	9	9	19	19	29	29
CDF per Landmark	1,142	1,142	1,058	1,058	1,037	1,037

Tabella X

Per Spirale, il peso dell'operazione di $V1$ è risultato ancora meno conveniente rispetto a Variazione Dimensioni al punto tale che quasi tutte le combinazioni di $V0$ erano migliori di quelle con $V1$ per tutte e tre le Griglie, inclusa quella Grande.

Come premesso, Spirale non ha avuto notevoli incrementi temporali nonostante la crescita del numero di Landmark. Questo è confermato dal fatto che l'aumento temporale dell'algoritmo è causato dal numero di iterazioni sulla frontiera come descritto da riga 15 dello pseudo-algoritmo e che la dimensione della Frontiera non dipende dal numero di Landmark.
In termini spaziali invece, Spirale ha portato un incremento a causa della Massima Profondità: aumentando il numero di Landmark, sono aumentate anche le iterazioni dell'algoritmo e di conseguentemente anche la profondità raggiunta.

3) Linea Spezzata: L'analisi della serie "Linea Spezzata" consente lo studio di un Cammino Minimo con numerose Celle di Frontiera. In confronto a Spirale, il numero di Landmark è simile ma il numero di Celle di Frontiera cresce come si evince dai dati riportati nelle tabelle statistiche delle sperimentazioni correnti.

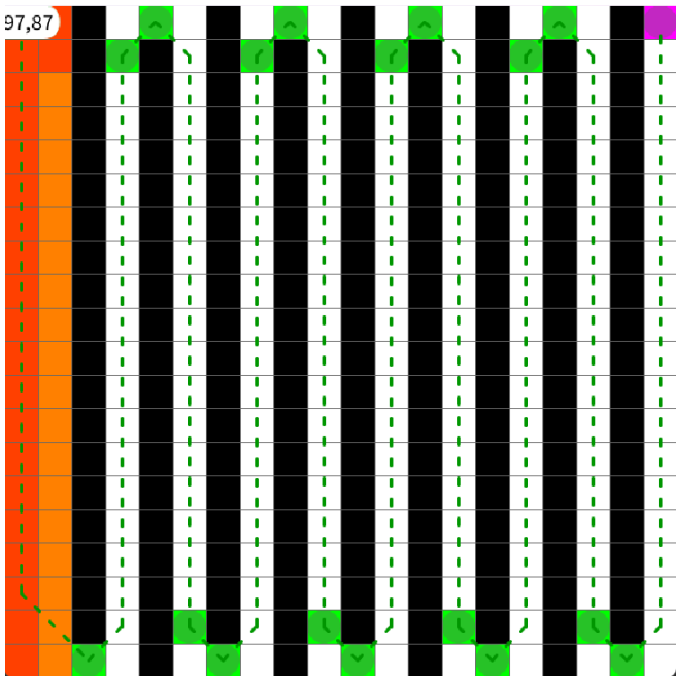


Figura 6

Le dimensioni utilizzate sono:

Griglia	w x h	Cache Hit	Landmark Intermedi
piccola	10x10	3	8
media	20x20	18	18
grande	30x30	48	38

Tabella XI

L'uso combinato di $CACHE$ e $CONDIZIONE\ \ RAFFORZATA$ ha diminuito il numero di volte di utilizzo della cache di 1.
Viene incrementata solamente la larghezza delle Griglie per verificare il comportamento dell'algoritmo all'aumentare di nuovi Landmark. La realizzazione avviene mediante uso di Ostacoli di tipo Barra Verticale (il che definisce il tipo delle Griglie usate per questi test). Le dimensioni inficiano dunque sul numero di Landmark presi in considerazione. Tutte le implementazioni analizzate hanno trovato un numero di Celle di Frontiera considerate variabile ma stessa lunghezza del Camino Minimo, a parità di Griglia scelta.
Vengono ora analizzate le differenti $ConfigurationMode$ usando $CompitoDueImpl.V0$:

L'uso di $PERFORMANCE$ si dimostra mediamente il migliore, in linea con quanto affermato precedentemente: l'uso di $CACHE$ ha portato ad un primo miglioramento dei risultati senza cache per Griglia Media e Griglia Grande e $CONDIZIONE\ \ RAFFORZTA$ e $SORTED\ FRONTIERA$ hanno poi ulteriormente giovato i risultati. Questa analisi evidenzia un aspetto già sottolineato: quando il numero di Celle di Frontiera comincia ad essere importante, la cache gioca un ruolo fondamentale per il Cammino Minimo e, grazie a Griglia Media e Griglia Grande, è possibile sottolineare questa componente che invece in Griglia Piccola non risalta, viste le ridotte dimensioni.
Per quanto riguarda invece le dimensioni, l'efficienza è dettata dalla massima profondità: per la Griglia Piccola, $NO\ \ SORTED\ \ FRONTIERA$, $PERFORMANCE$, $PERFORMANCE\ FULL$ sono i migliori con una massima profondità di 7 ed uno spazio stimato di 5,47 KB. Per la Griglia Media, è l'uso di cache a fare la differenza: tutte le implementazioni che la usano raggiungono una massima profondità di 15 ed uno spazio occupato stimato di 23,44 KB mentre tutte le altre hanno una profondità massima di 18 con uno spazio occupato di 28,13 KB. Anche per la Griglia Grande, la cache porta ad una massima profondità di 30 con 93,75 KB. Senza, si ha una massima profondità di 38 con 118,75 KB.

Modalità-Dimensione	Tempo esecuzione (ms)	Peso (KB)	Celle di Frontiera	Iterazioni Condizione	Massima Profondità	CDF per Landmark
DEFAULT						
piccola	5,748	7,03	93	0	9	11,625
media	5993,142	28,13	1021	0	18	56,722
grande	20363,952	118,75	4522220	0	38	119005,789
PERFORMANCE CACHE						
piccola	0,917	6,25	43	0	8	5,375
media	5,066	23,44	67	0	15	3,722
grande	17,050	93,75	157	0	30	4,132
PERFORMANCE CONDIZIONE RAFFORZATA						
piccola	9,678	6,25	39	12	8	4,875
media	11,158	28,13	936	170	18	52
grande	15753,232	118,75	961192	174762	38	25294,526
PERFORMANCE SORTED FRONTIERA						
piccola	1,166	7,03	93	0	9	11,625
media	14,772	28,13	1021	0	18	56,722
grande	20266,046	118,75	1048573	0	38	27594,026
PERFORMANCE NO CACHE						
piccola	0,469	6,25	30	3	8	3,75
media	12,439	28,13	936	85	18	52
grande	16378,440	118,75	961192	87381	38	25294,526
PERFORMANCE NO CONDIZIONE RAFFORZATA						
piccola	1,030	6,25	43	0	8	5,375
media	1,750	23,44	67	0	15	3,722
grande	8,061	93,75	157	0	30	4,132
PERFORMANCE NO SORTED FRONTIERA						
piccola	0,417	5,47	25	4	7	3,125
media	1,713	23,44	66	3	15	3,667
grande	8,998	93,75	156	3	30	4,105
PERFORMANCE						
piccola	0,509	5,47	22	2	7	2,75
media	1,219	23,44	66	2	15	3,667
grande	7,063	93,75	156	2	30	4,105
PERFORMANCE SVUOTA FRONTIERA						
piccola	0,784	7,03	67	0	9	8,375
media	2,272	28,13	171	0	18	9,5
grande	31,103	118,75	1439	0	38	37,868
PERFORMANCE FULL						
piccola	0,483	5,47	22	2	7	2,75
media	1,645	23,44	66	2	15	3,667
grande	5,626	93,75	156	2	30	4,105

Tabella XII

Si considerano ora i dati con $CompitoDueImpl.V1$:

Anche passando a $V1$, non ci sono miglioramenti sui tempi d'esecuzione ma sullo spazio occupato sì: per la Griglia Piccola, l'uso di cache ha permesso una riduzione del massimo livello di profondità passando da 8 a 7 e conseguentemente riducendo lo spazio di occupazione stimato: 5,47 KB. Per la Griglia Media, è solamente $PERFORMANCE\ NO\ SORTED\ FRONTIERA$ a raggiungere il miglior risultato: 21,88 KB con una profondità massima di 14. Questo significa che $V1$ ha usato cache e condizione rafforzata per ridurre la massima profondità ma con l'ordinamento delle Frontiere, questa riduzione non c'è, difatti $PERFORMANCE$ ha una massima profondità di 15. Questo è vero anche per Griglia Grande: $PERFORMANCE\ NO\ SORTED\ FRONTIERA$ raggiunge una massima profondità di 29 (uno in meno rispetto a $V0$) con un'occupazione spaziale di 90,63 KB.

modalità-dimensione	Cache hit	celle di frontieta	Iterazioni condizione	Tempo esecuzione (ms)	Spazio occupato (KB)	Massima Profondità	CDF per Landmark
DEFAULT							
piccola	0	39	0	0,809	6,25	8	4,875
media	0	1021	0	14,695	28,13	18	56,722
grande	0	1048573	0	13625,551	118,75	38	27594,026
PERFORMANCE CACHE							
piccola	4	23	0	0,834	5,47	7	2,875
media	14	55	0	3,719	23,44	15	3,056
grande	34	125	0	12,331	6,25	8	3,289
PERFORMANCE CONDIZIONE RAFFORZATA							
piccola	0	39	12	0,638	6,25	8	4,875
media	0	936	170	14,698	28,13	18	9,444
grande	0	961192	174762	12580,777	118,75	38	25294,526
PERFORMANCE SORTED FRONTIERA							
piccola	0	39	0	0,833	6,25	8	4,875
media	0	1021	0	13,212	8,13	18	56,722
grande	0	1048573	0	14227,789	118,75	8	27594,026
PERFORMANCE NO CACHE							
piccola	0	30	3	0,796	6,25	8	3,75
media	0	936	85	12,989	28,13	18	9,444
grande	0	961192	87381	13407,995	118,75	38	25294,526
PERFORMANCE NO CONDIZIONE RAFFORZATA							
piccola	4	23	0	0,971	5,47	7	2,875
media	1	55	0	3,303	23,44	15	3,056
grande	3	124	0	11,192	93,75	30	3,263
PERFORMANCE NO SORTED FRONTIERA							
piccola	4	23	4	0,939	5,47	7	2,875
media	14	54	2	3,116	21,88	14	3
grande	34	124	2	10,968	90,63	29	3,263
PERFORMANCE							
piccola	4	20	1	0,648	5,47	7	2,5
media	1	54	1	11,959	23,44	15	3
grande	3	124	1	7,930	93,75	30	2,263
PERFORMANCE SVUOTA FRONTIERA							
piccola	0	31	0	0,720	6,25	8	3,875
media	0	171	0	4,194	28,13	18	9,5
grande	0	1439	0	49,015	118,75	38	37,868
PERFORMANCE FULL							
piccola	4	20	1	0,550	5,47	7	2,5
media	14	54	1	1,814	23,44	15	3
grande	34	124	1	7,706	93,75	30	3,263

Tabella XIII

Viene ora effettuato un confronto tra le varie Griglie con differenti implementazioni del CompitoDue usando l'implementazione $PERFORMANCE\ \ FULL$ di CompitoTre:

A parità di combinazione, le statistiche volgono a favore di $V1$ che, grazie al numero di Celle di Frontiera inferiore, dovrebbe occupare meno tempo. Tuttavia, non è così e $V0$ supera di circa 2 millisecondi $V1$. Anche nel caso di Linea Spezzata, l'operazione di raggruppamento di $V1$ si dimostra non conveniente per l'algoritmo.

Rispetto alle sperimentazioni precedenti, i tempi d'esecuzione sono aumentati come aspettato, passando dall'ordine dei millisecondi fino ai 20 secondi e sono diminuiti in accordo con una diminuzione delle Celle di Frontiera portato dalle implementazioni con cache o svuotamento della Frontiera. È importante però notare che limitarsi ad affermare che il tempo d'esecuzione cresce con l'aumentare del solo numero di Celle di Frontiera non è corretto. Si rimanda alle sperimentazioni della Griglia Doppia Linea Spezzata e della Griglia Scacchiera per una più opportuna analisi.

	piccola		media		grande	
Metriche	V0	V1	V0	V1	V0	V1
Tempo (ms)	0,483	0,550	1,645	1,814	5,626	7,706
Cache hit	3	4	18	14	48	34
Celle di Frontiera	22	20	66	54	156	124
Iterazioni Condizione	2	1	2	1	2	1
Spazio (KB)	5,47	5,47	23,44	23,44	93,75	93,75
Massima Profondità	7	7	15	15	30	30
CDF per Landmark	2,75	2,5	3,667	3	4,105	3,263

Tabella XIV

In termini spaziali, i risultati sono molto simili alla sperimentazione della Griglia Spirale grazie al numero altrettanto simile della Massima Profondità che, in questo caso, non è influenzato dal numero di Celle di Frontiera considerate.

4) Doppia Linea Spezzata: La Griglia "Doppia Linea Spezzata" è simile alla Griglia "Linea Spezzata" ma anziché far uso di Ostacoli di tipo Barra Verticale, usa Ostacoli di tipo Barra Orizzontale per indurre il movimento seghettato del percorso, ed in aggiunta viene inserito un Ostacolo di tipo Barra Verticale al centro della Griglia.

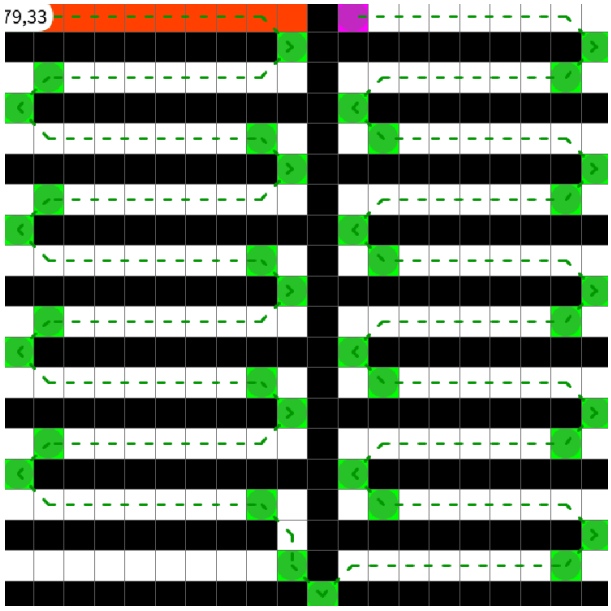


Figura 7

Le dimensioni utilizzate sono:

Griglia	Dimensioni	Cache hit	Iterazioni riga 16
piccola	10x10	13	0
media	20x20	62	0
grande	30x30	152	0

Tabella XV

Analogamente a Linea Spezzata, solo l'altezza della Griglia viene incrementata per consentire un numero di Landmark maggiore.
Lo scopo di questa Griglia è quello di verificare se il raddoppio dei Landmark porta ad un conseguente raddoppio dei tempi d'esecuzione. La risposta attesa è negativa.
Vengono ora analizzate le differenti $ConfigurationMode$ usando $CompitoDueImpl.V0$:

Modalità-Dimensione	Tempo esecuzione (ms)	Peso (KB)	CDF	Profondità	Landmark Intermedi	CDF per Landmark
DEFAULT						
piccola	5,748	14,06	706	18	18	39,222
media	5993,142	57,81	524797	37	37	13810,447
grande	Tempo Scaduto(15 m)	218,75	4522220	70	70	63693,239
PERFORMANCE CACHE						
piccola	0,917	11,72	57	15	18	3,167
media	5,066	45,31	220	29	37	5,789
grande	17,050	184,38	490	59	77	6,364
PERFORMANCE CONDIZIONE RAFFORZATA						
piccola	9,678	14,06	706	18	18	39,222
media	5126,738	57,81	481021	37	37	12658,447
grande	Tempo Scaduto(15 m)	218,75	4974490	70	70	71064,143
PERFORMANCE SORTED FRONTIERA						
piccola	5,831	14,06	706	18	18	39,222
media	5587,307	57,81	524797	37	37	13810,447
grande	Tempo Scaduto(15 m)	221,88	4177670	71	71	58840,423
PERFORMANCE NO CACHE						
piccola	5,884	14,06	651	18	18	36,167
media	4627,698	57,81	481021	37	37	13000,568
grande	Tempo Scaduto(15 m)	221,88	5315359	71	71	74864,211
PERFORMANCE NO CONDIZIONE RAFFORZATA						
piccola	0,841	12,50	57	15	18	3,167
media	4,889	46,88	220	30	37	5,946
grande	16,935	187,50	4522220	60	77	5870,129
PERFORMANCE NO SORTED FRONTIERA						
piccola	1,021	12,50	57	15	18	3,167
media	5,579	45,31	218	29	37	5,892
grande	17,051	184,38	490	59	77	6,636
PERFORMANCE						
piccola	0,822	12,50	56	16	18	3,111
media	4,629	46,88	218	30	37	5,892
grande	17,438	187,50	490	60	77	6,636
PERFORMANCE SVUOTA FRONTIERA						
piccola	7,274	14,06	201	18	18	11,167
media	29,991	57,81	1790	37	37	47,105
grande	458,631	240,63	21123	70	77	274,325
PERFORMANCE FULL						
piccola	1,136	12,50	57	16	18	3,167
media	4,848	46,88	220	30	37	5,946
grande	19,487	187,50	490	60	77	6,363

Tabella XVI

Considerando tutti i risultati, $PERFORMANCE$ è il metodo migliore sia per Griglia Piccola che per Griglia Grande mentre $PERFORMANCE\ NO\ CONDIZIONE\ RAFFORZATA$ è il migliore per Griglia Grande. Va tuttavia sottolineato come le implementazioni senza cache non risolvono la Griglia Grande entro il tempo limite ed ottengono risultati di gran lunga peggiori rispetto alle implementazioni con cache.
Nella Griglia Piccola, la sola cache porta la massima profondità a 15 con uno spazio di 11,72 KB mentre senza cache la profondità arriva a 18 con 14,06 KB. Va però sottolineato il comportamento della cache quando unito ad altre implementazioni: in combinazioni come $NO\ \ CONDIZIONE\ \ RAFFORZATA$ o $PERFORMANCE$, la massima profondità è 16 con 12,50 KB. Questo significa che per Doppio Dente di Sega la cache non solo consente di trovare il Cammino entro il tempo limite ma usare solamente $CACHE$ ha anche dei vantaggi a livello spaziale. Si considerano ora i dati con $CompitoDueImpl.V1$:

modalità-dimensione	Cache hit	Celle di Frontiera	Tempo esecuzione (ms)	Spazio occupato (KB)	Massima Profondità	Landmark Intermedi	CDF per Landmark
DEFAULT							
piccola	-	706	37,764	14,06	18	18	39,222
media	-	524627	3793,809	57,81	37	37	14179,108
grande	-	104001403	Tempo Scaduto(15 minuti)	221,88	71	71	1464808,493
PERFORMANCE CACHE							
piccola	12	51	1,901	11,72	15	18	2,833
media	31	117	5,502	45,31	29	37	3,162
grande	71	257	24,746	24,746	59	77	3,338
PERFORMANCE CONDIZIONE RAFFORZATA							
piccola	-	706	8,641	14,06	18	18	39,222
media	-	524627	3795,827	57,81	37	37	14179,108
grande	-	106392037	Tempo Scaduto(15 minuti)	221,88	71	71	1498479,394
PERFORMANCE SORTED FRONTIERA							
piccola	-	706	5,951	14,06	18	18	39,222
media	-	524627	3753,259	57,81	37	37	14179,108
grande	-	107778219	Tempo Scaduto(15 minuti)	225,00	72	72	1496919,708
PERFORMANCE NO CACHE							
piccola	-	706	6,103	14,06	18	18	39,222
media	-	524627	4036,234	57,81	37	37	14179,108
grande	-	106942538	Tempo Scaduto(15 minuti)	225,00	72	72	1485313,028
PERFORMANCE NO CONDIZIONE RAFFORZATA							
piccola	12	51	1,474	12,50	16	18	2,833
media	31	117	5,617	45,31	29	37	3,162
grande	71	257	20,602	184,38	59	77	3,338
PERFORMANCE NO SORTED FRONTIERA							
piccola	12	51	2,286	11,72	15	18	2,833
media	31	117	6,878	45,31	29	37	3,162
grande	71	257	22,549	184,38	59	77	3,338
PERFORMANCE							
piccola	12	51	1,780	12,50	16	18	2,833
media	31	117	4,869	45,31	29	37	3,162
grande	71	257	19,684	184,38	71	77	3,338
PERFORMANCE SVUOTA FRONTIERA							
piccola	-	201	2,066	14,06	18	18	11,167
media	-	1780	32,914	57,81	37	37	48,108
grande	-	21077	597,374	240,63	77	77	273,727
PERFORMANCE FULL							
piccola	12	51	1,580	12,50	16	18	2,833
media	31	117	6,601	45,31	29	37	3,162
grande	71	257	22,191	184,38	59	77	3,338

Tabella XVII

Per la Griglia Piccola, il cambio da $V0$ e $V1$ non porta alcun cambiamento mentre per Griglia Media e Griglia Grande la cache raggiunge un livello di profondità inferiore, riducendo lo spazio occupato: per la Griglia Media, si passa da una profondità di 30 a 29 con conseguente modifica dello spazio da 46,88 KB a 45,31 KB mentre per la Griglia Grande si passa da 60 a 59 ed uno spazio occupato da 187,50 KB a 184,38 KB. Viene ora effettuato un confronto tra le varie Griglie con differenti implementazioni del CompitoDue usando l'implementazione $PERFORMANCE\ FULL$ di CompitoTre: come per il caso di Linea Spezzata, la riduzione delle statistiche d'esecuzione si può notare ma, anche per questo caso, l'operazione di raggruppamento non si dimostra conveniente per il tempo d'esecuzione ma offre comunque uno spazio d'occupazione lievemente inferiore.

Come anticipato nelle premesse, il tempo d'esecuzione rispetto a Linea Spezzata non è raddoppiato ma ha invece avuto una crescita non lineare in termini temporali. Questo è dovuto al numero di Celle di Frontiera che sono cresciute esponenzialmente rispetto alla Griglia Linea Spezzata.
In termini spaziali invece si può riscontrare un raddoppio delle dimensioni proprio grazie all'approssimativo raddoppio della Massima Profondità raggiunta dall'algoritmo.

	piccola		media		grande	
Metriche	V0	V1	V0	V1	V0	V1
Tempo (ms)	1,136	1,580	4,848	6,601	23,828	22,191
Cache hit	13	12	62	31	152	71
Celle di Frontiera	57	51	220	117	490	257
Iterazioni Condizione	0	0	0	0	0	0
Spazio (KB)	12,50	12,50	46,88	45,31	187,50	184,38
Massima Profondità	16	16	30	29	60	59
CDF per Landmark	3,167	2,833	5,946	3,162	6,363	3,338

Tabella XVIII

5) Scacchiera: La Griglia Scacchiera verifica il comportamento dell'algoritmo quando il Cammino Minimo è intenzionalmente privo di molte Celle di Frontiera da considerare ma con numerosi Landmark. In particolare, si vuole considerare un percorso simile a quello della sola Linea Spezzata per risaltare la non esatta correlazione tra le sole Celle di Frontiera ed il tempo d'esecuzione. In termini spaziali, è prevedibile un aumento dello spazio occupato a causa dei numerosi Landmark.

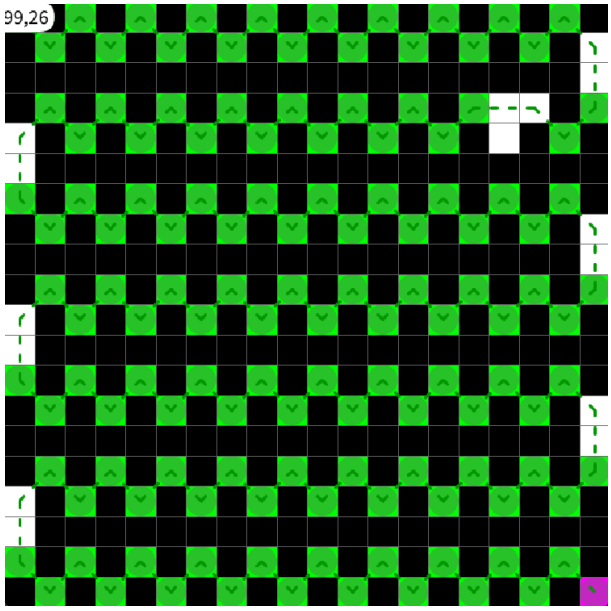


Figura 8

Le dimensioni utilizzate sono:

	Griglia		
Proprietà	piccola	media	grande
Dimensione ($w \times h$)	10x10	20x20	30x30
Spazio occupato	47,66	204,69	3,09 MB
Celle di frontiera	64	132	508
Condizioni riga 16	0	0	0
Massima profondità	61	131	506
Landmark Intermedi	61	131	506
Cache hit	1	0	0
CDF per Landmark	1,049	1,008	1,004

Tabella XIX

Le condizioni di miglioramento attivate sono 0 per tutte le combinazioni e tutte le Griglie tranne che per Griglia Grande con $PERFORMANCE\ CONDIZIONE\ RAFFORZATA$ che invece ha usato la condizione di miglioramento 1 volta sia con $CompitoDueImpl.V0$ che con $CompitoDueImpl.V1$.

Vengono ora analizzate le differenti $ConfigurationMode$ usando $CompitoDueImpl.V0$:

Modalità-Dimensione	Tempo esecuzione (ms)
DEFAULT	
piccola	0,936
media	2,087
grande	16,075
PERFORMANCE CACHE	
piccola	1,281
media	3,901
grande	42,181
PERFORMANCE CONDIZIONE RAFFORZATA	
piccola	1,037
media	1,941
grande	16,639
PERFORMANCE SORTED FRONTIERA	
piccola	1,103
media	1,666
grande	16,170
PERFORMANCE NO CACHE	
piccola	0,986
media	1,921
grande	18,275
PERFORMANCE NO CONDIZIONE RAFFORZATA	
piccola	1,323
media	3,581
grande	45,658
PERFORMANCE NO SORTED FRONTIERA	
piccola	1,380
media	3,463
grande	42,154
PERFORMANCE	
piccola	1,515
media	4,535
grande	50,747
PERFORMANCE SVUOTA FRONTIERA	
piccola	1,104
media	2,066
grande	18,432
PERFORMANCE FULL	
piccola	1,601
media	4,465
grande	45,007

Tabella XX

Scacchiera ha dunque evidenziato come nonostante una Griglia con numerosi ostacoli, la loro disposizione ha formato un Cammino Minimo che non lasciasse spazio a variazioni di percorso, portando a poche Celle di Frontiera da considerare in rispetto al numero di Landmark effettivamente usati. Per questi casi, il percorso è relativamente semplice e dunque non usare implementazioni aggiuntive si dimostra efficace.
Si considerano ora i dati con $CompitoDueImpl.V1$:

modalità-dimensione	Cache hit	Celle di Frontiera	Tempo esecuzione (ms)	Spazio occupato (KB)	Massima Profondità
DEFAULT					
piccola	-	62	1,587	47,66	61
media	-	130	4,293	204,69	131
grande	-	1048573	60,849	3090,00	506
PERFORMANCE CACHE					
piccola	1	23	2,161	47,66	61
media	0	130	5,573	204,69	131
grande	0	506	82,367	3090,00	506
PERFORMANCE CONDIZIONE RAFFORZATA					
piccola	-	84	1,694	47,66	61
media	-	130	4,255	61,513	131
grande	-	506	61,513	3090,00	506
PERFORMANCE SORTED FRONTIERA					
piccola	-	84	1,788	47,66	61
media	-	130	4,448	204,69	131
grande	-	506	59,100	3090,00	506
PERFORMANCE NO CACHE					
piccola	-	84	1,613	47,66	61
media	-	130	4,809	204,69	131
grande	-	506	65,660	3090,00	506
PERFORMANCE NO CONDIZIONE RAFFORZATA					
piccola	1	62	1,851	47,66	61
media	0	130	5,830	204,69	131
grande	0	506	79,515	3090,00	506
PERFORMANCE NO SORTED FRONTIERA					
piccola	1	62	1,822	47,66	61
media	0	130	5,811	204,69	131
grande	0	506	80,411	3090,00	506
PERFORMANCE					
piccola	1	62	1,942	46,88	60
media	0	130	5,817	204,69	131
grande	0	506	81,104	3090,00	506
PERFORMANCE SVUOTA FRONTIERA					
piccola	0	62	1,701	47,66	61
media	0	130	4,952	204,69	131
grande	0	506	65,217	3090,00 506	
PERFORMANCE FULL					
piccola	1	62	1,827	46,88	60
media	0	130	5,924	204,69	131
grande	0	506	85,179	3090,00	506

Tabella XXI

Contrariamente alle Griglie precedenti, in termini di tempo $V1$ è peggiore in tutti i casi e con tutte le combinazioni rispetto a $V0$. Scacchiera evidenzia dunque il peso dell'operazione di $CompitoDueImpl.V1$: non è possibile escludere alcuna Cella data la composizione delle Griglie e questa operazione grava sul peso ad ogni incremento della profondità e Scacchiera raggiunge alti livelli di profondità proprio a causa dei numerosi Landmark. Per questa specifica Griglia quindi $V1$ non si dimostra efficace.

Viene ora effettuato un confronto tra le Griglie Grandi con differenti implementazioni del CompitoDue usando l'implementazione $PERFORMANCE\ \ FULL$ di CompitoTre:

	piccola		media		grande	
Metriche	V0	V1	V0	V1	V0	V1
Tempo (ms)	1,601	1,827	4,465	5,924	45,007	85,179
Cache hit	0	1	0	0	0	0
Celle di Frontiera	64	62	132	130	507	506
Iterazioni Condizione	0	0	0	0	0	0
Spazio (KB)	47,66	46,88	204,69	204,69	187,50	3090
Massima Profondità	61	60	131	131	507	506
CDF per Landmark	1,032	1,016	1,008	0,992	1,004	1

Tabella XXII

Anche dal punto di vista esecutivo, la differenza tra $V0$ e $V1$ risiede in 2 Celle di Frontiera in meno ma le operazioni di $V1$ hanno portato al quasi raddoppio del tempo necessario per il Cammino Minimo.

Per sottolineare la dimostrazione di Scacchiera, si possono

confrontare le implementazioni $DEFAULT$ con $V0$ di Linea Spezzata per la Griglia Grande e quella di Scacchiera per la Griglia Piccola: il numero di Celle di Frontiera per la prima sono di 706 mentre per la seconda sono 506. Nonostante queste misure siano relativamente vicine, i tempi d'esecuzione sono invece molto distanti: 5,748 millisecondi per Linea Spezzata e 0,936 millisecondi per Scacchiera. La differenza chiave dietro questa differenza sta invece nel numero di Celle di Frontiera per Landmark: in Linea Spezzata, il valore ammonta a 39,222 mentre in Scacchiera è 1,004. Come si ammoniva nelle conclusioni sulla Linea Spezzata, non è il solo numero di Celle di Frontiera ad aumentare il tempo d'esecuzione ma è il numero di Celle di Frontiera che ogni Landmark deve considerare. La spiegazione risiede nell'algoritmo, specificatamente nel ciclo `for` di riga 15 dello pseudo-codice: se ogni Landmark ha una sola Cella di Frontiera, il ciclo `for` svolgerà poche iterazioni per ogni Landmark. Se invece ogni Landmark ha un cospicuo numero di Celle di Frontiera, ogni iterazione appesantirà il tempo totale notevolmente. Nel caso di Scacchiera, la media è di circa una Cella di Frontiera per Landmark, da qui i ridotti tempi d'esecuzione.
Dal punto di vista spaziale, come previsto, i numeri sono aumentati a causa della Massima Profondità cresciuta notevolmente per via del numero di Landmark portando per la prima volta lo spazio occupato all'ordine dei Mega byte.

6) Tipo Griglia: Data la classificazione del Tipo di una Griglia, sono state condotte analisi su Griglie con tutte le possibili combinazioni di ostacoli.

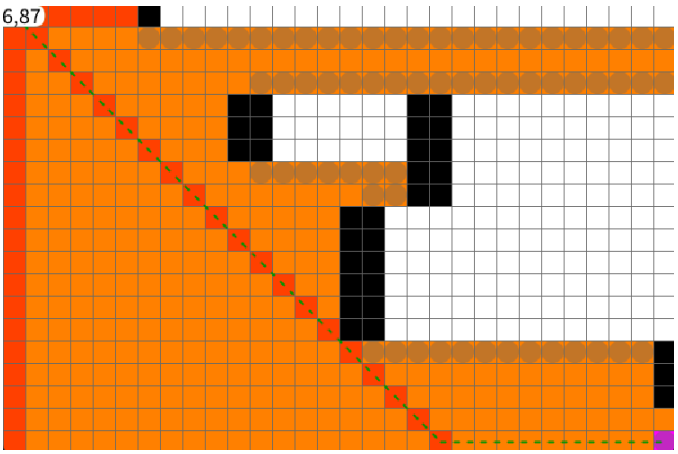


Figura 9

Le dimensioni delle Griglie sono uguali per tutti i tipi e fisse a 30x20. La generazione è avvenuta tramite automazione via codice ed utilizzando l'implementazione del Compito Uno con un generatore pseudo-casuale con un Random Seed fisso a 42 per tutte le Griglie. Per formalità, nell'analisi di queste griglie viene esplicitamente definito il tipo della Griglia nonostante la lettura di una Griglia che, di norma, dovrebbe avere un tipo personalizzato. Per convenzione, l'Origine è presa nel punto più in alto a sinistra, ossia (0,0) mentre la Destinazione nel punto più in basso a destra, ossia (29,19).
Dato che i tipi di Ostacoli sono 8, tutte le possibili combinazioni e quindi i tipi di Griglia sono 2^8, ossia 256 tipi di Griglia. La generazione avviene usando un numero fisso di numero massimo di Ostacoli (il cui valore cambia in base al tipo d'Ostacolo) e combinandoli tra di essi alternando tra il numero massimo stabilito e assenza del tipo d'ostacolo, generando così le 256 Griglie. È tuttavia importante considerare che la convenzione sulla scelta di Origine e Destinazione ha scartato alcuni di questi tipi: data la posizione della Destinazione, la presenza dell'Ostacolo Delimitatore Verticale o Delimitatore Orizzontale ha impedito all'algoritmo di trovare Cammini che potessero raggiungere la Destinazione vista l'impossibilità di attraversare Ostacoli. Ciononostante, l'algoritmo ha reso nota questa difficoltà ed i risultati sono stati salvati negli appositi file.
Ciò che emerge da queste sperimentazioni è che $PERFORMANCE$ con $CompitoTreImpl.V0$ è il metodo prediletto per ottenere il più basso tempo d'esecuzione per la quasi totalità delle Griglie mentre, invece, i casi peggiori sono raggiunti in maniera equilibrata dalle altre implementazioni. Nei casi in cui $PERFORMANCE$ è la migliore delle implementazioni, il più alto valore di tempo d'esecuzione è 8.1 millisecondi, eccezion fatta per la Griglia di tipo 50 (0b00110010) che ha un tempo minimo d'esecuzione pari a 5.2 millisecondi, raggiunto da $PERFORMANCE\ NO\ CONDIZIONE\ RAFFORZATA$ con $CompitoTreImpl.V0$. In dettaglio, il tempo medio di $PERFORMANCE$ è di circa 15 millisecondi contro i quasi 12 millisecondi di $PERFORMANCE\ NO\ CONDIZIONE\ RAFFORZATA$. Per comprendere le ragioni dietro questo specifico caso, viene mostrata la Griglia 50 con Origine e Destinazione. Questa Griglia ha la Destinazione entro il Contesto dell'Origine, difatti l'esecuzione non considera alcuna Cella di Frontiera trovandosi nel caso base dell'algoritmo. Il miglioramento di $PERFORMANCE\ NO\ CONDIZIONE\ RAFFORZATA$ non è dunque dovuto a livelli di prestazioni effettivamente migliori ma è da attribuirsi alle prestazioni del dispositivo in uso per le sperimentazioni e della JVM usata per esse.

Un'altra Griglia che invece presenta molte Celle di Frontiera iniziali è la Griglia di tipo 20.

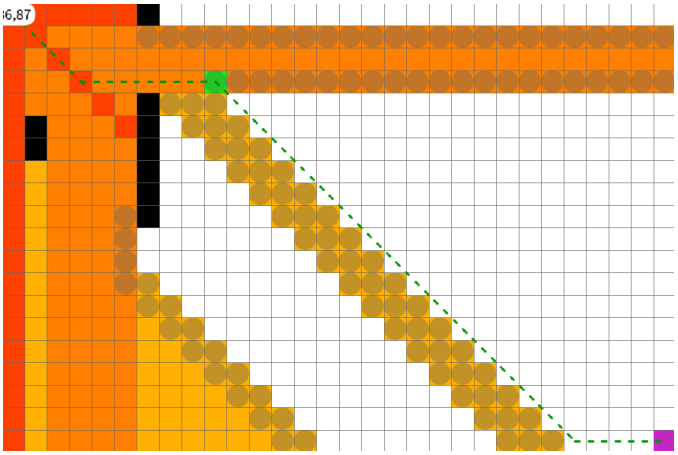


Figura 10

La Destinazione non si trova nel Contesto dell'Origine e sono presenti numerose Celle di Frontiera iniziali. In questo caso, il miglior tempo d'esecuzione è stato raggiunto da $PERFORMANCE\ \ FULL$ con $V0$ con 289630 nanosecondi. Seguono poi $PERFORMANCE\ CONDIZIONE\ RAFFORZATA$ con $V1$ in 299960 nanosecondi e $PERFORMANCE\ NO\ CACHE$ con $V1$ in 313910 nanosecondi. La carenza di ostacoli nella Griglia ha prodotto un Cammino Minimo molto semplice rispetto ad altre sperimentazioni antecedenti, ma ha ugualmente sottolineato l'importanza delle implementazioni aggiuntive, soprattutto se si considera che sempre per la Griglia di tipo 20 $DEFAULT$ con $V0$ ha richiesto 65,039 millisecondi.

Un'altra Griglia degna d'attenzione è la tipo 52:

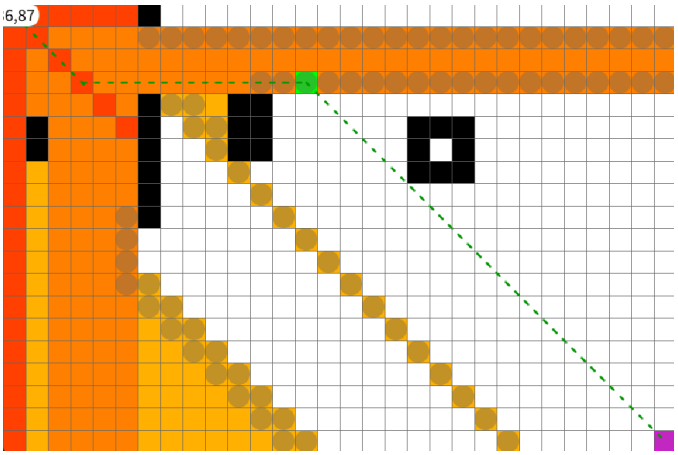


Figura 11

Nonostante la sua semplicità, è l'unica griglia che richiede svariati minuti tranne che per le implementazioni con $CONDIZIONE\ RAFFORZATA$, sottolineando l'utilità di essa in Griglia dalle ridotte dimensioni ma dalle numerose Celle di Frontiera per Landmark: sebbene la Griglia abbia solamente 3 Landmark, $DEFAULT$ considera 6077186 Celle di Frontiera, $PERFORMANCE\ CACHE$ ne considera 4642978 e con l'aggiunta di $SORTED\ FRONTIERA$ si riduce a 2841642 ma con l'uso della sola $CONDIZIONE\ \ RAFFORZATA$ e con eventuali altre aggiunte si considerano solamente 80 Celle di Frontiera. Anche senza $CONDIZIONE\ RAFFORZATA$, un altro modo per abbattere i tempi d'esecuzione della Griglia tipo 52 è l'uso del Compito Due $V1$ che, grazie all'uso della sola regione che contiene la Destinazione, ha permesso la riduzione di Celle di Frontiera non rilevanti per il Cammino Minimo.
Considerando dunque le Griglie analizzate, è possibile affermare che la modalità $PERFORMANCE$ sia la migliore. È tuttavia necessario puntualizzare che queste sperimentazioni utilizzano istanze in cui le Griglie vengono generate con pseudo-casualità: al contrario delle precedenti sperimentazioni, non c'è la volontà di verificare un comportamento specifico dell'algoritmo, sia esso la risposta ad un numeroso quantitativo di Celle di Frontiera da considerare, un elevato numero di Landmark o considerevoli dimensioni della Griglia stessa. Quando invece si considera l'utilizzo di $CompitoDueImpl.V1$, ci sono variazioni circa la miglior prestazione: alcune Griglie tendono a preferire il Compito Due $V1$ con qualsiasi implementazione di CompitoTre, altre ancora invece è l'implementazione di CompitoTre a fare la differenza mentre CompitoDue non sembra portare alcun particolare cambiamento. Tuttavia, dal punto di vista dell'occupazione spaziale, $V1$ risulta sempre la migliore: talvolta, non vi è differenza tra $CompitoDueImpl.V0$ e $CompitoDueImp.V1$ mentre altre volte $V1$ occupa la metà dello spazio rispetto a $V0$, altre volte ancora invece $V1$ occupa un quinto dello spazio rispetto a $V0$.
Alla luce di ciò, è importante sottolineare che tuttavia a volte $V1$ è invece peggiore di $V0$: questo vale per quelle Griglie dal Cammino relativamente "semplice" con poche Celle di Frontiera la cui esecuzione è relativamente breve in termini di tempo e la cui differenza tra la miglior combinazione con $V0$ e la miglior combinazione di $V1$ risiede in qualche migliaio di nanosecondi.
Dunque, è possibile infine concludere che $V1$ offra il miglior compromesso in termini di prestazioni: $V1$ è l'implementazione di CompitoDue migliore per quanto riguarda l'occupazione spaziale ed offre ottimi risultati (spesso anche i migliori) per quanto riguarda il tempo richiesto per l'esecuzione.

7) Variazione Ostacoli: Un'ultima sperimentazione viene eseguita su una Griglia più fedele ad una reale implementazione.

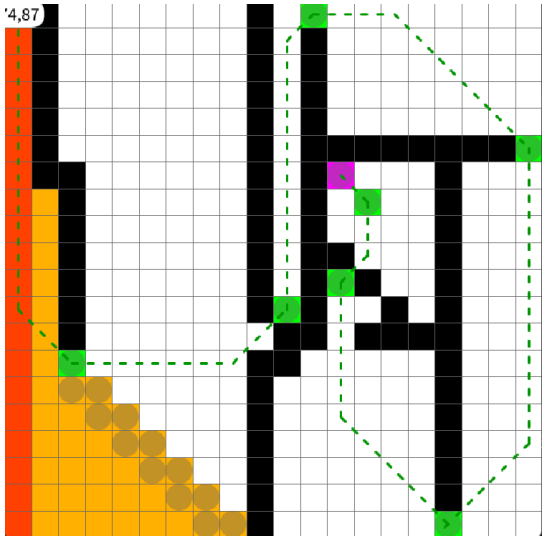


Figura 12

Vengono ora analizzate le differenti $ConfigurationMode$ usando $CompitoDueImpl.V0$:

Questa Griglia richiede 8 Landmark Intermedi per raggiungere la destinazione ed il Cammino Minimo ha una lunghezza di $74,870$. La Griglia ha una dimensione 20x20.

Data la varietà della Griglia, è possibile sottolineare non solo l'efficacia della cache per il tempo ma anche l'incremento di prestazioni spaziali con $SVUOTA\ FRONTIERA$: mentre tutte le combinazioni raggiungono una profondità massima di 13 o 14, $PERFORMANCE\ FULL$ e $PERFORMANCE\ SVUOTA\ FRONTIERA$ arrivano fino a 12, raggiungendo lo spazio occupato minore: 18,75 KB.

Nella pagina successiva vengono considerati i dati con $CompitoDueImpl.V1$.

Tempo esecuzione (s)	Cache Hit	Spazio (KB)	Celle di Frontiera	Condizione riga 16	Profondità	CDF per Landmark
DEFAULT						
6,399	-	21,88	435311	0	14	54413,875
PERFORMANCE CACHE						
0,068	1423	21,88	3504	0	14	438
PERFORMANCE CONDIZIONE RAFFORZATA						
3,538	-	20,31	392473	160792	13	49059,125
PERFORMANCE SORTED FRONTIERA						
6,789	-	21,88	435311	0	14	54413,875
PERFORMANCE NO CACHE						
2,899	-	20,31	337113	0	13	42139,125
PERFORMANCE NO CONDIZIONE RAFFORZATA						
0,070	1423	21,88	3504	0	14	438
PERFORMANCE NO SORTED FRONTIERA						
0,048	1409	20,31	3371	546	13	68,25
PERFORMANCE						
0,051	1339	20,31	3026	504	13	63
PERFORMANCE SVUOTA FRONTIERA						
1,854	-	18,75	121818	0	12	15227,25
PERFORMANCE FULL						
0,04	1120	18,75	2483	225	12	28,125

Tabella XXIII

Cache hit	Celle di Frontiera	Iterazioni condizione	Tempo esecuzione (ms)	Spazio occupato (KB)	Massima Profondità	CDF per Landmark
DEFAULT						
-	23581	0	138,202	15,63	10	2947,625
PERFORMANCE CACHE						
199	282	0	5,428	12,50	8	35,25
PERFORMANCE CONDIZIONE RAFFORZATA						
-	22621	8342	100,577	15,63	10	2827,625
PERFORMANCE SORTED FRONTIERA						
-	23581	0	142,211	15,63	10	2947,625
PERFORMANCE NO CACHE						
-	20381	7382	107,306	15,63	10	2547,625
PERFORMANCE NO CONDIZIONE RAFFORZATA						
199	282	0	4,206	14,06	9	35,25
PERFORMANCE NO SORTED FRONTIERA						
176	272	25	3,679	12,50	8	34
PERFORMANCE						
166	263	29	3,539	12,50	8	32,875
PERFORMANCE SVUOTA FRONTIERA						
0	12535	0	89,687	15,63	10	1566,875
PERFORMANCE FULL						
166	263	29	3,585	12,50	8	32,875

Tabella XXIV

Grazie a Variazione Ostacoli, si può sottolineare come $V1$ abbia un impatto positivo sulle prestazioni sia in termini di tempo che in termini di spazio: il miglior tempo in assoluto viene da $PERFORMANCE$ (molto vicino comunque a $PERFORMANCE\ FULL$) con 3,539 millisecondi. Lo stesso $PERFORMANCE$ con $V0$ veniva eseguito in 51,069 millisecondi. In termini di spazio, le combinazioni con $V1$ migliori sono quelle che fanno uso di cache con una massima profondità di 8 ed uno spazio occupato di 12,50 KB. Viene ora effettuato un confronto con differenti implementazioni del CompitoDue usando l'implementazione $PERFORMANCE\ FULL$ di CompitoTre:

Metriche	V0	V1
Tempo (ms)	43,839	3,585
Cache hit	1120	166
Celle di Frontiera	2483	263
Iterazioni Condizione	225	29
Spazio (KB)	18,75	12,50
Massima Profondità	12	8

Tabella XXV

In questo caso, $V1$ porta grosse differenze: le statistiche non spaziali sono circa un decimo rispetto a quelle di $V0$.
La Griglia Variazione Ostacoli presenta diverse caratteristiche che nelle Griglie precedenti venivano accentuate per esaltare i diversi comportamenti dell'algoritmo: vi è infatti un considerevole numero di Celle di Frontiera per Landmark con un contenuto numero di Landmark ma ha in più il fatto che la strutturazione della Griglia favorisce la riduzione della Frontiera considerata ad ogni iterazione, spiegando quindi il particolare beneficio temporale ottenuto dall'uso di $CompitoDueImpl.V1$. Con queste considerazioni, si conferma che il numero di Celle di Frontiera per Landmark è un aspetto cruciale nelle prestazioni temporali.
Dal punto di vista delle prestazioni spaziali, la Massima Profondità è influenzata anche dalla ricerca del Cammino Minimo e l'esclusione di parte della Frontiera in questo ha favorito un raggiungimento di un livello di profondità inferiore ed una conseguente riduzione dello spazio occupato, come si evince dal confronto tra le implementazioni di $V0$ e $V1$.

D. Risultati

A seguito di tutte le sperimentazioni, è possibile genericamente affermare che in caso di un Cammino Minimo "semplice", ossia con poche Celle di Frontiera ed una Griglia di basse dimensioni, l'uso di $DEFAULT$, ossia senza ulteriori aggiunte implementative, è principalmente la scelta migliore. Quando invece la Griglia comincia a crescere in dimensioni e nel numero di Celle di Frontiera da considerare, optare per $PERFORMANCE$ o al più $PERFORMANCE\ NO\ SORTED\ FRONTIERA$ potrebbe essere la scelta migliore, in particolar modo è l'uso di cache che fornisce le migliorie più significative.
Per quanto riguarda invece le implementazioni di CompitoDue, è possibile affermare che $V1$ aumenta d'efficacia con l'aumentare delle dimensioni, in lieve misura, ma soprattutto con degli "spazi aperti" ampi, ossia quelle Griglie che hanno un'ampia area di Celle non Ostacolo, con abbastanza Celle di Frontiera da considerare e la cui Destinazione non risiede nel Contesto o Complemento dell'Origine. Tuttavia, l'aumentare del livello di profondità tende a ridurre questo vantaggio ma, in caso le condizioni sopra restino vere, $V1$ sarà comunque superiore in termini di tempo e spazio.
Quando si vogliono invece considerare le prestazioni temporali, è fondamentale considerare il numero di Celle di Frontiera per Landmark: un numero elevato porta ad una crescita esponenziale dei tempi. In questi casi, l'uso di cache è la soluzione preferita, seguita dall'uso di svuotamento di Frontiera.
Per l'aspetto spaziale bisogna invece considerare la Massima Profondità, assumendo che le dimensioni della Griglia non siano variabili. Per mediare il valore di Profondità, è opportuno utilizzare $CompitoDueImpl.V1$ che riduce efficacemente questo valore.

E. Verifiche correttezza

Per verificare in maniera più approfondita ed estesa la correttezza, vengono condotte numerose verifiche di correttezza su diverse griglie con 4 dimensioni differenti e generate con le modalità descritte in CompitoUno. Le caratteristiche sono:

- Larghezza: 30, Altezza: 20, Seed: 42;
- Larghezza: 100, Altezza: 75, Seed: 21;
- Larghezza: 175, Altezza: 200, Seed: 5;
- Larghezza: 250, Altezza: 140, Seed: 12.

Ogni Griglia ha un diverso numero d'ostacoli (e conseguentemente un tipo diverso) in base alle dimensioni. Ogni Griglia è stata analizzata 10 volte ed ognuna di queste analisi viene eseguita una volta per ogni combinazione di CompitoDue e CompitoTre le quali estraevano una Origine ed una Destinazione casuale ammessa nella Griglia. Per facilità di sperimentazione, il tempo limite viene abbassato a 5 minuti. Quando una delle due esecuzioni (da Origine a Destinazione o da Destinazione a Origine) non viene completata entro il tempo limite o non è possibile raggiungere la Destinazione, quella combinazione di Origine e Destinazione viene scartata.
Alla fine delle sperimentazioni, tutti i casi si dimostrano corretti con medesima lunghezza del Cammino Minimo e numero di Landmark.

F. Grandi dimensioni

Per poter verificare il comportamento dell'algoritmo meno nel dettaglio di Variazione Dimensioni ma con una grandezza maggiore, sono state condotte delle sperimentazioni a parte con Griglie dalle seguenti dimensioni:

- Larghezza: 100, Altezza: 100
- Larghezza: 500, Altezza: 500
- Larghezza: 1000, Altezza: 1000
- Larghezza: 2000, Altezza: 2000

In questo caso, il tempo limite è di 20 minuti e le Griglie sono generate usando l'implementazione del CompitoUno. Il tipo delle Griglie è sempre lo stesso ed il numero degli Ostacoli cresce in proporzione diretta con le dimensioni della Griglia. Per rapidità nelle sperimentazioni, non vengono prese in considerazione tutte le combinazioni di CompitoTre e CompitoDue ma solo $DEFAULT$ e $PERFORMANCE\ FULL$ con $CompitoDueImpl.V0$ e $CompitoDueImpl.V1$.

A causa dell'implementazione ricorsiva di V1 e lo heap space di Java limitato, $CompitoDueImpl.V1$ non è in grado di operare con Griglie di notevoli dimensioni. Dai risultati delle Sperimentazioni si può concludere che la Griglia con dimensioni 500x500 viene utilizzata correttamente mentre la Griglia 1000x1000 termina lo spazio a disposizione nello heap causando un `OutOfMemoryError`.

Usando $V0$, per la Griglia 1000x1000 e la Griglia 2000x2000 $DEFAULT$ esaurisce il tempo d'esecuzione disponibile mentre $PERFORMANCE\ FULL$ trova il Cammino Minimo in rispettivamente circa 3 minuti e 3 secondi. La disparità di tempi è dovuto alla massima profondità: nonostante entrambe le Griglie usino 1 solo Landmark tra Origine e Destinazione, la Griglia 1000x1000 raggiunge una Massima Profondità di 4 mentre la Griglia 2000x2000 di 2. Le Griglie 100x100 e 500x500 invece vengono eseguite da entrambe le implementazioni di CompitoTre e richiedono rispettivamente circa 1 millisecondo e circa 50 millisecondi. Dato che la Destinazione è presente nella CHIUSURA dell'Origine, le modalità non dimostrano differenze statistiche per entrambe le griglie.

6. Conclusioni

In questa relazione è stata documentata l'implementazione di un sistema per la navigazione su griglia. Il lavoro svolto ha permesso di sviluppare una soluzione completa, partendo dalla generazione di ambienti di prova con ostacoli di diverse tipologie fino alla valutazione delle prestazioni dell'algoritmo di ricerca del cammino.

L'implementazione dell'algoritmo di ricerca del cammino ha prodotto percorsi validi e corretti per le griglie considerate in tutte le modalità di esecuzione disponibili.

È stato possibile osservare come il numero di Celle di Frontiera per Landmark influenzi direttamente il tempo di calcolo e la memoria utilizzata, evidenziando le sfide pratiche di tali problemi.

A. Limitazioni

1) Limite profondità ricorsione: Al fine di verificare le limitazioni dell'algoritmo causate dalla Java Virtual Machine (JVM), è stata usata la Griglia Scacchiera presente con la creazione del CompitoUno: grazie all'elevato numero di Landmark dovuto alla struttura della Griglie stessa, si è arrivati alle dimensioni massime di 61x60 o 60x65, con circa 1100 Landmark. Con questi valori, la JVM solleva uno `StackOverflowError` a causa dell'elevato numero di ricorsioni che esaurisce lo Stack.

2) Collisioni delle chiavi hash: L'implementazione di un sistema di caching basato su funzioni hash nel Compito Tre ha dimostrato notevoli benefici prestazionali. Nonostante i vantaggi, è fondamentale considerare la potenziale occorrenza di collisioni nella generazione delle chiavi. Sebbene non siano stati rilevati casi specifici durante le sperimentazioni, il rischio teorico non può essere escluso. Una collisione, se si verificasse, potrebbe compromettere la correttezza algoritmica e portare a risultati inattesi o errati. Questo rischio rappresenta una limitazione intrinseca del sistema. La funzione hash utilizzata: `Objects.hash(..)` [4] sfrutta l'`hashCode` della Classe `Point2D`. Alcune collisioni con `Point2D` sono:

- (-47, -33) e (-48, -32) : hash = -2021130240
- (-47, -32) e (-48, -33) : hash = -2021163008
- (-47, 32) e (-48, 33) : hash = 126320640
- (93, 66) e (30, 69) : hash = -1981235200

3) Limite spaziale della cache: Il problema del raggiungimento del limite di memoria Heap quando la cache è attiva, specialmente su Griglie con un elevato numero di cammini, rappresenta una limitazione significativa. Questo fenomeno, che si manifesta con un'eccezione di tipo `OutOfMemoryError`, è dovuto all'uso massivo di memoria da parte della cache per memorizzare i risultati intermedi. Sebbene la cache migliori le prestazioni evitando ricalcoli, il suo approccio "aggressivo" all'uso della memoria la rende inadatta per scenari con elevato numero di Cammini, compromettendo la scalabilità del sistema.

4) Limite spaziale dell'algoritmo: Il problema del raggiungimento del limite di memoria Heap non è esclusivo dell'uso della cache, ma può manifestarsi anche a causa della dimensione della griglia stessa. Quando l'algoritmo crea e salva in memoria un numero eccessivo di copie della Griglia, viene superata la capacità della memoria RAM disponibile. Sebbene il meccanismo sia simile a quello osservato con la cache, in questo caso la causa principale è la complessità spaziale dell'algoritmo di base, non l'ottimizzazione del caching.

B. Sviluppi futuri

Sebbene l'attuale implementazione sia efficiente, ci sono diverse direzioni per miglioramenti futuri.

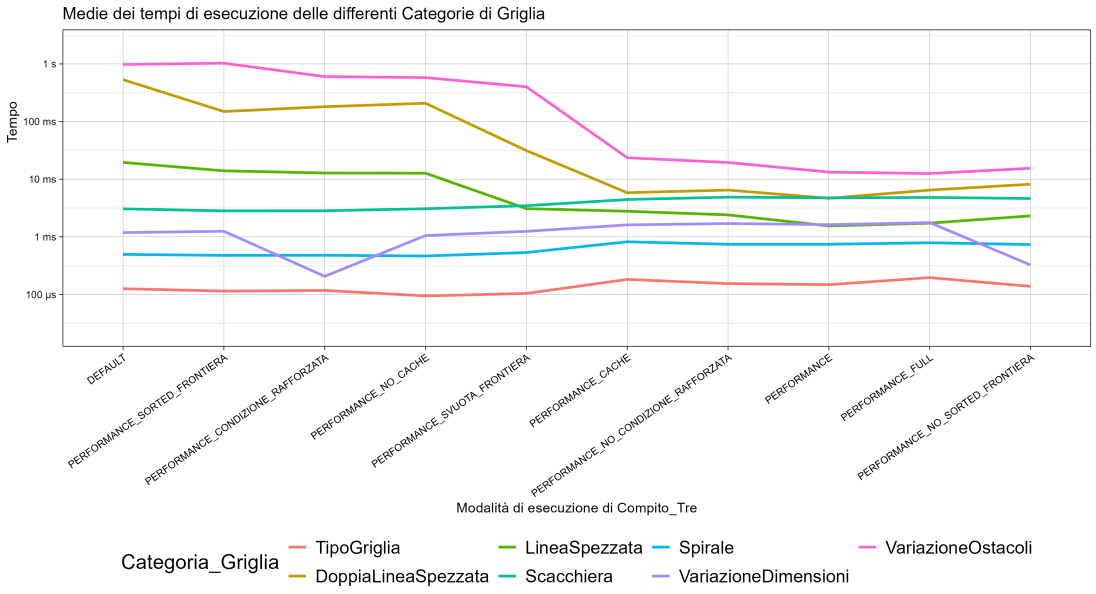


Figura 13: Tempi d'esecuzione al variare delle configurazioni del Compito Tre

1) Limite profondità ricorsione: La limitazione della profondità di ricorsione con un elevato numero di chiamate può essere risolta convertendo l'algoritmo ricorsivo in una versione iterativa. Questo approccio elimina la dipendenza dallo Stack delle chiamate del sistema operativo, sostituendolo con una struttura dati esplicita (come uno stack gestito manualmente o una coda) per tracciare lo stato dell'esecuzione. Un'implementazione iterativa non solo previene i problemi di overflow dello Stack, ma può anche offrire un maggiore controllo sull'allocazione della memoria e sulle performance, rendendo l'algoritmo scalabile anche per Cammini molto lunghi e complessi. Questo permetterebbe l'utilizzo dell'implementazione alternativa del Compito Due tramite il doppio dizionario e i vantaggi ad esso associato anche per griglie di grandi dimensioni.

2) Collisioni delle chiavi hash: Per mitigare il rischio di collisioni nella funzione hashCode di Point2D, è possibile implementare una delle seguenti strategie:

- **Rafforzamento della funzione hash:** sviluppare una funzione hash personalizzata che consideri non solo le coordinate del punto, ma anche altre proprietà uniche del nodo (es. un identificatore univoco). Un'alternativa più sicura consiste nell'utilizzare una funzione hash crittografica (e.g. SHA-256 [5]), sebbene possa comportare un overhead computazionale.
- **Gestione esplicita delle collisioni:** è possibile gestire il caso in cui si verifichino le collisioni. Invece di sovrascrivere il valore, la cache potrebbe usare una struttura dati per la gestione delle collisioni, come liste concatenate o tabelle hash a indirizzamento aperto, garantendo così la correttezza del dato recuperato.

3) Limite spaziale della cache: Il problema del raggiungimento del limite di spazio disponibile nello Heap con la cache attiva, può essere affrontato con diverse soluzioni:

- **Implementazione di una cache a capacità limitata:** usare strategie di espulsione come Least Recently Used (LRU) o Least Frequently Used (LFU). Questo significa che, una volta raggiunto un certo limite di memoria, la cache rimuoverà automaticamente gli elementi più vecchi o meno usati per liberare spazio.
- **Serializzazione dei dati su disco:** per Griglie particolarmente grandi, dove anche una cache ottimizzata non è sufficiente, si può considerare la serializzazione di una parte dei dati della cache su disco. Questa tecnica, pur introducendo una latenza aggiuntiva, permette di gestire insiemi di dati che superano la capacità della memoria RAM disponibile.

4) Limite spaziale dell'algoritmo: Non essendo possibile usare le sopracitate strategie di espulsione per la cancellazione delle Griglie non utili al passo considerato, il problema può essere risolto in diversi modi:

- **Serializzazione selettiva su file:** Una soluzione coinvolge la serializzazione delle Griglie che non sono immediatamente necessarie, mantenendo in memoria solo quelle attive o di recente utilizzo. Questo approccio ibrido, che combina la memoria volatile (RAM) con quella persistente (disco), permette di gestire insiemi di dati molto più grandi;
- **Compressore di Griglie**: Invece di salvare l'intera Griglia, questa soluzione coinvolge il salvataggio delle sole differenze tra una Griglia e la successiva. Se la maggior parte delle Griglie sono simili, questo metodo è in grado di ridurre notevolmente l'ingombro di memoria. Per ricostruire una Griglia specifica, è sufficiente partire da uno stato "base" e applicare la sequenza di differenze registrate;
- **Strutture dati sparse**: Se la Griglia contiene principalmente celle vuote (è una Griglia "sparsa"), memorizzare

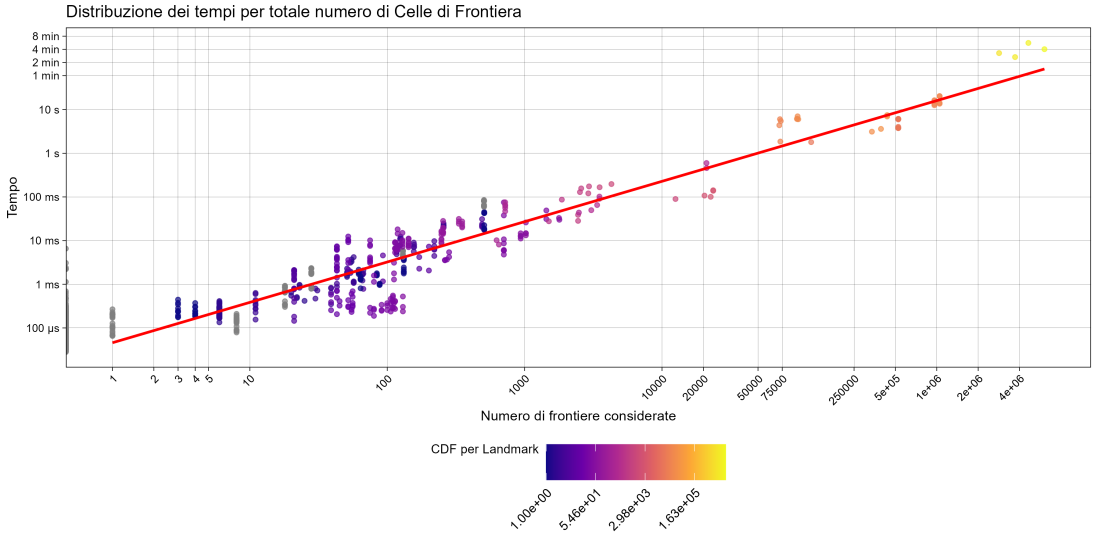


Figura 14: Andamento dei tempi delle sperimentazioni al variare del numero di Celle di Frontiera considerate, con mappa dei colori rappresentate la quantità di CDF per Landmark

l'intera struttura in un array bidimensionale può essere inefficiente. È possibile optare per strutture dati che salvano solo le posizioni delle Celle non vuote, come un dizionario che associa le coordinate (x,y) all'Ostacolo. Questa soluzione è in grado di ridurre significativamente lo spazio occupato.

7. Utilizzo del codice

A. Configurazione del JSON

Il file di configurazione è da usarsi in formato JSON. Nel codice sono già presenti esempi del file di configurazione strutturati come segue:

```
config.json
├─ width
├─ height
├─ randomSeed
├─ maxOstacoli:
│   ├─ SEMPLICE
│   ├─ AGGLOMERATO
│   ├─ BARRA_VERTICALE
│   ├─ BARRA_ORIZZONTALE
│   ├─ BARRA_DIAGONALE
│   ├─ ZONA_CHIUSA
│   ├─ DELIMITATORE_VERTICALE
│   └─ DELIMITATORE_ORIZZONTALE
└─ disposizione:
    ├─ STANDARD
    ├─ VARIAZIONE_DIMENSIONI
    ├─ SPIRALE
    ├─ LINEA_SPEZZATA
    ├─ DOPPIA_LINEA_SPEZZATA
    └─ SCACCHIERA
```

Figura 15

- **width**: Indica la larghezza della Griglia;
- **height**: Indica l'altezza della Griglia;
- **randomSeed**: Indica il *seed* utilizzato per la generazione pseudo-casuale degli Ostacoli;
- **maxOstacoli**: È un JSON nidificato che contiene come chiave il nome di tutti i tipi di Ostacoli (eccezion fatta per "PERSONALIZZATO") e come valore il numero massimo di Ostacoli da inserire;
- **disposizione**: È un JSON nidificato che indica la formazione degli Ostacoli nella Griglia. Per attivare una formazione, è necessario impostare `true` come valore della formazione desiderata. La lettura delle formazioni avviene in ordine, dunque solo la prima formazione a cui il programma legge `true` viene presa in considerazione. Nel caso in cui tutte le formazioni abbiano valore `false` o il JSON nidificato risulta assente, viene utilizzata la modalità STANDARD;

B. Main

Nel package `matteo` è presente:

- La classe `Main`, utilizzabile per eseguire l'algoritmo.
- Il file `matteo/config.json` che presenta una struttura simile al `config.json` del Compito Uno, ma con l'aggiunta di configurazione per l'esecuzione del Compito Tre.

Di seguito sono riportati i parametri di configurazione del progetto presenti nel file `matteo/config.json`, con dei valori di esempio. Tutti i parametri qui riportati sono nell'oggetto nidificato `impostazioniMain`.

```
config.json
├─ ...
└─ impostazioniMain:
    ├─ tempoLimite: 30
    ├─ timeUnit: "MINUTES"
    ├─ origineX: 0
    ├─ origineY: 0
    ├─ destinazioneX: 10
    ├─ destinazioneY: 10
    ├─ compitoDueModalita: "V0"
    ├─ compitoTreModalitaFlags:
    │   ├─ DEBUG: false
    │   ├─ MONITOR_ENABLED: true
    │   ├─ SORTED_FRONTIERA: false
    │   ├─ CONDIZIONE_RAFFORZATA: false
    │   ├─ CACHE_ENABLED: false
    │   └─ SVUOTA_FRONTIERA: false
    └─ tipoRiassunto: "VERBOSE"
```

Figura 16

impostazioniMain: Contiene i parametri principali di configurazione.

- **tempoLimite**: limite massimo di tempo.
- **timeUnit**: unità di misura del tempo.
- **origineX**, **origineY**: coordinate di Origine.
- **destinazioneX**, **destinazioneY**: coordinate di Destinazione.
- **compitoDueModalita**: modalità di esecuzione scelta per il Compito Due.
- **compitoTreModalitaFlags**: attiva/disattiva specifiche opzioni di configurazione.
  - **DEBUG**: abilita messaggi di debug.
  - **MONITOR_ENABLED**: abilita monitoraggio.
  - **SORTED_FRONTIERA**: ordina la Frontiera.
  - **CONDIZIONE_RAFFORZATA**: applica condizione rafforzata di potatura.
  - **CACHE_ENABLED**: abilita cache.
  - **SVUOTA_FRONTIERA**: svuota la Frontiera quando possibile.
- **tipoRiassunto**: Tipo di riassunto desiderato.

C. AppletMain

Il package `main` contiene la classe `AppletMain`, la quale offre una rappresentazione grafica interattiva del progetto sfruttando la libreria open-source di Processing [6].

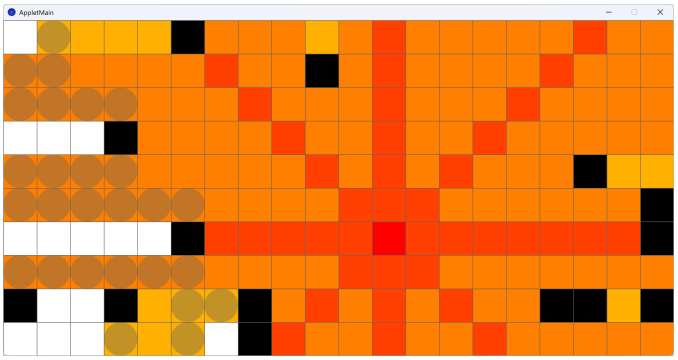


Figura 17: Finestra principale

Figura 18: Finestra secondaria delle impostazioni configurabili a runtime

Tasto	Azione
TAB	Apri/chiudi la finestra delle impostazioni
C	Ripulisci la griglia mantenendo le celle non navigabili
CTRL+C	Ripulisci la griglia rimuovendo le celle non navigabili
P	Stampa a video la griglia con gli stati delle celle
SHIFT+P	Stampa a video la griglia in formato semplice
R	Stampa a video il Riassunto *VERBOSE* di camminoMin
I	Interrompi l'esecuzione di camminoMin
M	Stampa coordinate del mouse nella griglia
U	Visualizza le regioni colorate
F	Stampa a video le celle di frontiera del contesto attuale

Tabella XXVI: Comandi da tastiera AppletMain

Evento	Azione
Click destro	Posiziona l'Origine
Click sinistro	Posizione la Destinazione
Click centrale	Inverte la navigabilità di una Cella
Trascinamento destro	Posiziona Celle non navigabili
Trascinamento sinistro	Rimuove Celle non navigabili

Tabella XXVII: Azioni del mouse AppletMain

```
config.json
├─ ...
├─ load:
│  ├─ path: "griglie.int.json"
│  └─ name: "test4"
├─ loadThis: [
│      [ , ,1, ],
│      [1, ,1, ],
│      [1,1, , ],
│  ]
└─ applet:
   ├─ width: 1200
   ├─ height: 600
   ├─ showText: false
   ├─ pixelDensity: 1
   └─ palette
      ├─ ORIGINE:      "0xff0000"
      ├─ REGINA:       "0xff4000"
      ├─ CONTESTO:     "0xff8000"
      ├─ COMPLEMENTO:  "0xffb000"
      ├─ FRONTIERA:    "0x004000"
      ├─ VUOTA:        "0xffffff"
      ├─ OSTACOLO:     "0x000000"
      ├─ DESTINAZIONE:"0xff00ff"
      └─ LANDMARK:     "0x00ff00"
```

Figura 19

- **load**: Parametro opzionale, se presente carica la griglia col nome `name` dal file JSON presente nel percorso; `path`
- **loadThis**: Parametro opzionale, se presente carica la griglia memorizzata a tale attributo come lista di liste di interi;
- **width**: Indica la larghezza massima in pixel della finestra;
- **height**: Indica l'altezza massima in pixel della finestra;
- **showText**: booleano per disattivare le scritte degli stati di ogni cella;
- **pixelDensity**: Densità di pixel per aumentare la risoluzione delle scritte, se supportato;
- **palette**: Dizionario che associa ad ogni `StatoCella` un colore in esadecimale.

RIFERIMENTI BIBLIOGRAFICI

[1] https://docs.oracle.com/javase/specs/jvms/se8/html/jvms-2.html.
[2] https://it.wikipedia.org/wiki/Ricerca_in_profondità.
[3] https://docs.oracle.com/javase/8/docs/api/java/util/Arrays.html.
[4] https://docs.oracle.com/javase/8/docs/api/java/util/Objects.html.
[5] https://it.wikipedia.org/wiki/Secure_Hash_Algorithm.
[6] https://processing.org/reference/.