Elaborato di Algoritmi e Strutture Dati

Stefano Fontana

Anno Accademico 2022/23
Appello di Settembre

Sommario

Questo documento contiene la spiegazione del codice e i risultati relativi alla sua analisi necessari per superare l'insegnamento di Algoritmi e Strutture Dati nell'anno accademico 2022/23.

Indice

1 Testo dell'elaborato

Di seguito viene riportato un riassunto del testo dell'elaborato e le relative consegne richieste.

1.1 Definizioni

1.1.1 Griglia

Nell'ambito di questo elaborato, viene definita *Griglia* oppure *Campo* uno spazio bidimensionale di dimensioni definite e di coordinate intere all'interno del quale possono muoversi uno o più agenti.

Ogni mossa cardinale, dove lo spostamento da un tempo t a un tempo $t+1$ avviene cambiando al più una coordinata, ha peso 1, mentre qualora vi fosse uno spostamento diagonale questo avrà peso $\sqrt{2}$ [4, p. 3].

Una cella c viene detta *attraversabile* quando un agente può effettuare una mossa dalla propria posizione alla cella c, mentre non attraversabile in caso contrario. È necessario notare che una cella può essere attraversabile per un istante temporale t ma non per un istante precedente o successivo t' in quanto più entità possono muoversi all'interno della griglia ed esse non possono coesistere all'interno di una stessa cella.

È necessario notare come ogni istanza di una griglia induca un grafo orientato e pesato. Possiamo con ciò definire la griglia come nella definizione 1 utilizzando la nozione di vicinanza definita dalla definizione 2.

$$
\begin{aligned}
&\mathbf{Griglia}(\mathbf{width}, \mathbf{height}) = (\mathbf{V}, \mathbf{E}, \mathbf{w}) \ t.c. \\
&\mathbf{V} = \big\{(x, y) \in \mathbb{N} \times \mathbb{N} : x < width \wedge y < height \wedge (x, y) \ is \ not \ obstacle\big\} \\
&\mathbf{E} = \big\{((x_1, y_1), (x_2, y_2)) \in \mathbf{V} \times \mathbf{V} : (x_1, y_1) \ vicina \ (x_2, y_2)\big\} \\
&\mathbf{w}((x_1, y_1), (x_2, y_2)) \to \mathbb{R} := \begin{cases} 1 \ se \ x_1 - x_2 = 0 \vee y_1 - y_2 = 0 \\ \sqrt{2} \ altrimenti \end{cases}
\end{aligned}
\tag{1}
$$

$$
\begin{aligned}
&(x_1, y_1), (x_2, y_2) \in \mathbf{V} \\
&(x_1, y_1) \ vicina \ (x_2, y_2) \iff \mid x_1 - x_2 \mid < 2 \wedge \mid y_1 - y_2 \mid < 2
\end{aligned}
\tag{2}
$$

Come è possibile notare dalla definizione 2 una cella è considerata vicina a essa stessa, indi per cui nel grafo indotto dalla griglia per ogni cella attraversabile è presente un cappio il cui peso è pari a 1 [4, p. 5].

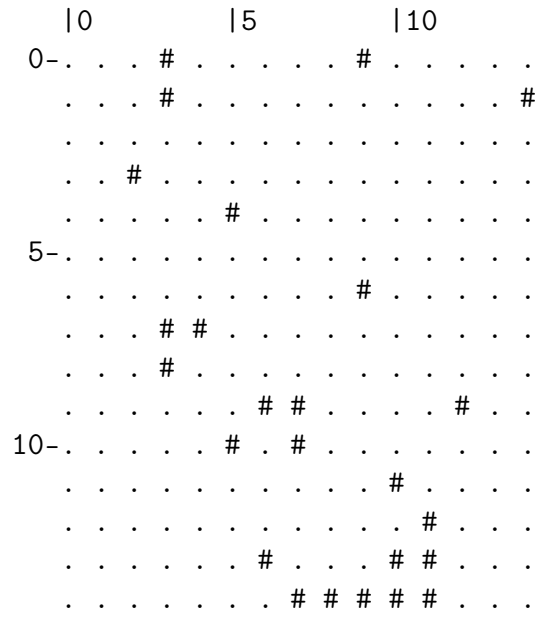


Figura 1: Esempio di rappresentazione di una griglia 15x15 con 25 ostacoli

Notazione

La notazione utilizzata all'interno di questo documento al fine di rappresentare una griglia viene mostrata alla figura 1. Essa consiste in una rappresentazione per carattere dove il simbolo . indica la presenza di una cella vuota attraversabile e il simbolo # rappresenta un ostacolo. È inoltre possibile osservare come il sistema di numerazione ponga l'origine delle coordinate nell'angolo in alto a sinistra e inizi la numerazione da 0.

1.1.2 Percorso

Definiamo percorso una sequenza π di archi sul grafo indotto definito alla definizione 1. Gli elementi della sequenza prendono dunque il nome di *mosse*.

Si assume che ogni mossa effettuata da un agente venga eseguita in tempo unitario e che essa sia atomica, ovvero che l'agente a sia collocato nella cella c al tempo t e successivamente alla mossa esso si ritrovi nella cella c' *vicina a* c nell'istante di tempo $t + 1$.

$$\begin{aligned} &\pi = \left\{v \in \mathbf{V}\right\}_{1..n}, n \in [0, t_{max}] : \forall i \in [1, |\pi|)\pi_i \; vicina \; \pi_{i+1} \\ &w(\pi) = \sum_{k=1}^{|\pi|-1} w(\pi_k, \pi_{k+1}) \end{aligned} \tag{3}$$

Si deduce dalla definizione 3 che un percorso avrà durata pari a $|\pi| - 1$ in quanto vengono considerate le transizioni di una lista di lunghezza $|\pi|$. Inoltre si evidenzia che un agente a ogni istante di tempo t può effettuare le seguenti mosse:

- Non cambiare cella ($\pi_t = \pi_{t+1}$), ovvero percorrere il cappio presente sulla cella. A questa mossa si farà riferimento nominandola "*mossa wait*".
- Spostarsi attraverso un arco appartenente a $\mathbf{E}$ in una cella adiacente alla cella di partenza.

Dalla la presenza di molteplici agenti all'interno della medesima griglia, i quali si muovono contemporaneamente, nascono i vincoli definiti nei successivi paragrafi.

Una cella (x, y) non può essere occupata da due agenti distinti nello stesso istante t, ovvero

$$\forall \pi, \pi' \in \Pi, \pi_t \neq \pi'_t \forall t \in [1, t_{max}] \tag{4}$$

dove Π rappresenta l'insieme di tutti i percorsi di tutti gli agenti. Inoltre non è possibile che due agenti si scambino di posto se questi occupano due celle adiacenti. Ne segue

$$\forall \pi, \pi' \in \Pi, \pi_t = u, \pi'_t = v \; u, v \in \mathbf{V} \implies \pi_t \neq v \vee \pi'_t \neq u \forall t \in [1, t_{max}) \tag{5}$$

laddove t_{max} denota l'ultimo istante temporale possibile.

Qualora vengano a meno le condizioni definite nelle definizioni 4 e 5 il percorso entra in *collisione*. Si noti che dalle due definizioni due agenti possono seguirsi e che è possibile effettuare un incrocio diagonale dei percorsi.

Notazione

La notazione utilizzata all'interno di questo documento al fine di rappresentare un percorso viene mostrata alla figura 2. Essa sfrutta la notazione della griglia definita nella sezione 1.1.1 e rapresenta con un valore incrementale la posizione di un agente all'interno di una cella. Nel caso un agente rimanga più tempo nella stessa cella, effettuando quindi mosse *wait*, verrà mostrato solo il tempo iniziale della permanenza provocando un "salto" all'interno della sequenza. Se venissero mostrati più percorsi essi cambieranno stile di notazione. In caso di ambiguità potrà essere usata una notazione esplicita riportando più volte la rappresentazione della griglia come in figura 3.

Nella figura 2 vengono mostrati i percorsi (5, 9)→(6, 8)→(7, 8)→(8, 8)→(8, 8)→(9, 9)→(9, 9)→(10, 10) e (10, 5)→(11, 4)→(12, 4)→(13, 5), mentre nella figura 3 vengono mostrate le sequenze (0, 0)→(1, 1)→(1, 1)→(2, 1) e (2, 0)→(2, 1)→(1, 2)→(0, 2).

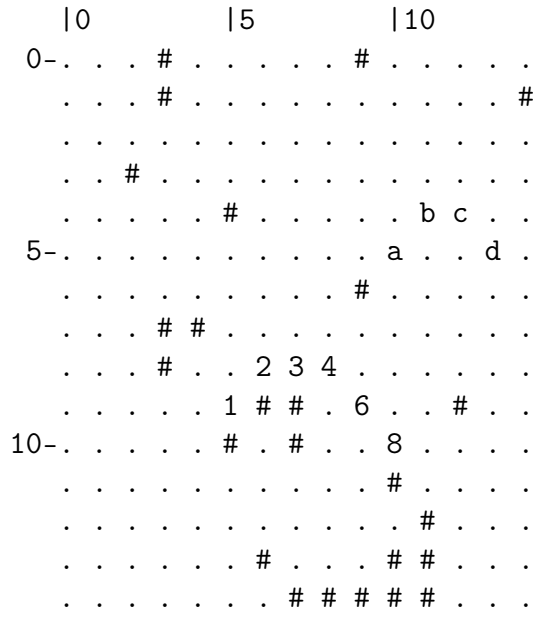
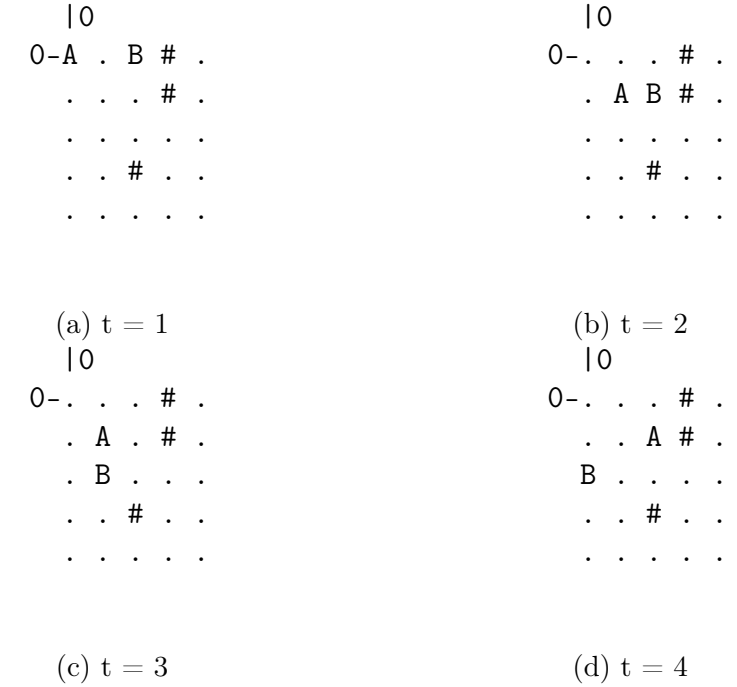


Figura 2: Esempio di rappresentazione di due percorsi



(a) t = 1

(b) t = 2

(c) t = 3

(d) t = 4

Figura 3: Esempio di rappresentazione di due percorsi in modo esteso

1.1.3 Istanza

Definiamo *Istanza* del problema "Path Finding for an entry Agent" la tupla definita dalla definizione 6

$$PF4EA = ((\mathbf{V}, \mathbf{E}, \mathbf{w}), \Pi, init, goal, t_{max}) \tag{6}$$

dove Π è l'insieme di tutti i percorsi degli n agenti,
$init \in \mathbf{V}$ è una cella dove viene situata la partenza del percorso dell'entry agent tale che

$$\forall \pi \in \Pi, \pi_0 \neq \ init$$

e $goal \in \mathbf{V}$ sia la cella finale del percorso tale che

$$\forall \pi \in \Pi, \pi_{t_{max}} \neq \ goal$$

e t_{max} l'istante temporale massimo.

La soluzione del problema consiste quindi in un percorso

$$\bar{\pi} \ t.c. \ |\bar{\pi}| \leq t_{max} \wedge \bar{\pi}_0 = \ init \ \wedge \bar{\pi}_{t_{max}} = \ goal$$

dove $w(\bar{\pi}) = \min w(\bar{\Pi}_{init,goal})$ con $\bar{\Pi}_{init,goal}$ l'insieme di tutti i percorsi validi aventi inizio in *init* e terminazione in *goal* che rispetti le condizioni 3, 4 e 5.

1.2 Elaborato

1.2.1 Generatore di istanze

Si chiede di progettare e realizzare un generatore di griglie, che dati opportuni parametri costruisca la griglia e posizioni in modo semi casuale gli ostacoli entro la stessa. Ogni invocazione deve portare alla creazione della griglia.

In alternativa è possibile progettare un generatore di istanze che, a ogni invocazione generi tutti i parametri dell'istanze del problema PF4EA con tutti i percorsi degli agenti definiti in modo semi casuale nel rispetto delle definizioni 3, 4 e 5.

La seconda richiesta sussume la prima in quanto la griglia è parte dell'output della seconda richiesta.

All'utente dei generatori di cui sopra deve essere consentito di configurare dei parametri, ritenuti significativi, che guidino i generatori stessi nelle scelte non totalmente casuali.

1.2.2 Algoritmo risolvente

Dato l'algoritmo risolutivo presente nell'appendice A progettare e implementare l'algoritmo `ReconstructPath` scrivendone lo pseudocodice il cui compito è quello di ricostruire il percorso a costo minimo secondo le definizioni precedenti.

Sviluppare quindi l'algoritmo `ReachGoal` partendo da quello definito nell'appendice A nella versione priva delle righe 19-22 prestando particolare attenzione alla definizione di strutture dati appropriate tali da limitare l'occupazione di memoria all'aumentare dei seguenti parametri:

- Dimensione della griglia
- Orizzonte temporale t_{max}
- Lunghezza della soluzione

Utilizzare una funzione *h(a, b)* appropriata (e.g. Distanza metrica, Manhattan Distance, Lunghezza del cammino rilassato senza agenti, Costo del cammino minimo senza agenti).

1.2.3 Strategia alternativa

Implementare le righe mancanti dell'algoritmo disponibile nell'appendiceA. Tale soluzione consiste, a ogni iterazione dell'algoritmo `ReachGoal`, nel valutare il percorso rilassato (il quale non varia al procedere dell'algoritmo risolutivo), privo di interazioni con gli agenti e, se quest'ultimo è valido, allora ritorna direttamente interrompendo le iterazione della ricerca A*.

1.3 Consegne e Richieste

Si conduca una sperimentazione sia con il generatore di griglie o di istanze di PF4EA (cfr. sez. 1.2.1 sia con il risolutore con entrambe le implementazioni date dalle sezioni 1.2.2 e 1.2.3)

Un fine della sperimentazione è quello di registrare e valutare criticamente le prestazioni temporali e spaziali delle prove condotte.

Per conseguire l'obiettivo di descrivere il comportamento degli algoritmi nell'ambito delle prove, è positivo che l'applicazione sviluppata, per ogni istanza elaborata da ciascuna versione di ReachGoal, produca, oltre al percorso dell'entry agent, alcune informazioni supplementari, quali il numero di stati espansi, il numero di stati inseriti nella lista `Closed`, la lunghezza e il costo del percorso calcolato, nonché il numero di mosse wait contenute nel percorso calcolato.

Si sottoponga ogni volta a entrambe le versioni di ReachGoal realizzate la stessa istanza d'ingresso, confrontando i costi dei percorsi prodotti in uscita (per verificare la correttezza) nonché le prestazioni (spaziali e temporali) riscontrate.

Per ogni istanza d'ingresso, l'applicazione software sviluppata deve produrre automaticamente un riassunto in cui siano registrate le caratteristiche:

- dell'istanza (natura dei percorsi contenuti entro l'istanza, che chiarisca se tali percorsi sono stati creati in modo pseudocasuale o attraverso più invocazioni di ReachGoal, dimensione della griglia, numero/percentuale di celle attraversabili presenti nella griglia, numero di agenti, lunghezza dei percorsi degli agenti preesistenti, valore dell'orizzonte temporale t_{max} e valori dei parametri ritenuti necessari alla sperimentazione

- dell'elaborazione compiuta da ciascuna delle due versioni dell'algoritmo ReachGoal (sez. 1.2.2 e 1.2.3) in termini di soluzione prodotta, numero di stati visitati, euristica adottata per la prima versione di ReachGoal, lunghezza e costo del percorso prodotto in uscita oppure registrazione del fatto che l'elaborazione è terminata con un fallimento o non è terminata entro il tempo stabilito o è terminata perché richiedeva una occupazione di memoria superiore al limite stabilito, tempo di esecuzione, eventualmente anche massima occupazione di memoria

2 Elaborato

L'elaborato è stato scritto in "Rust" [5] in quanto l'impiego di questa tecnologia è un esperimento personale relativamente alla capacità di interiorizzare un nuovo linguaggio con semantiche e logiche di programmazione diverse e particolari. Trattasi di un linguaggio compilato molto performante con una forte considerazione verso la gestione efficiente e sicura della memoria.

In particolare è doveroso sottolineare come in questo linguaggio sia pressoché impossibile creare un eseguibile o libreria con fuoriuscite di memoria o bachi legati agli accessi nella stessa.

Ulteriore motivazione di questa scelta proviene dalla voglia di discostarsi dai più comuni linguaggi di programmazione utilizzati permettendo così di apprendere nuove metodologie di lavoro dalle quali prendere spunto nella vita lavorativa.

2.1 Generatore di istanze

In questa sezione vengono illustrati gli algoritmi e le metodologie utilizzate nella generazione delle istanze. Questa funzione viene svolta tramite l'applicativo `instance_gen` prendendo in ingresso un file di configurazione in formato YAML [2] oppure una serie di parametri da linea di comando.

Il risultato dell'elaborazione viene inviato nello stream di uscita (`stdout`) in modo che possa essere reindirizzato a un file.

Il ruolo del generatore di istanze è di notevole importanza in quanto esso permette di ridurre significativamente l'occupazione di memoria e le tempistiche durante la risoluzione dell'istanza.

2.1.1 Generazione della griglia

Il problema di generazione della griglia viene affrontato mediante un approccio matematico basato su una funzione pseudocasuale in due variabili. L'idea è di determinare la posizione degli ostacoli all'interno della griglia tramite la valutazione di una funzione secondo il metodo definito a seguire.

Per arrivare a questa soluzione non è possibile utilizzare una singola funzione pseudocasuale tratta da un generatore poiché, a parità di configurazione iniziale, la sequenza è uguale ma elementi singoli della sequenza non possono essere computati senza computare tutti i loro predecessori.

L'utilizzo di un singolo generatore pseudocasuale comporterebbe la necessità di calcolare sequenzialmente tutti gli elementi precedenti per poter valutare la funzione in una cella di coordinate date. Più in particolare, data una cella (x, y), il calcolo della funzione richiederebbe la computazione

di $f(0,0)...f(w,0), f(0,1)...f(0,y), f(1,y)...f(x,y)$. Anche supponendo un accesso con probabilità uniforme all'interno della griglia — ovvero che le variabili x e y abbiano una distribuzione uniforme all'interno dell'intervallo di appartenenza — si ottiene che il numero medio di computazioni della funzione converge a

$$x \in [1,w], y \in [1,h],\ x,y \in \mathbb{N}$$
$$E[x] = \sum_{x=1}^{w} \frac{1}{w} x = \frac{1}{w} \sum_{x=1}^{w} x = \frac{1}{w} \frac{w(w+1)}{2} = \frac{w+1}{2}$$
analogo per y

$$E[sequential] = E[y] * w + E[x] = \frac{hw+w}{2} + \frac{w+1}{2} \approx \Theta(\frac{w \cdot h}{2}) \quad (7)$$
$$E[hilbert]^1 = E[x]E[y] = \frac{(w+1)(h+1)}{4} \approx \Theta(\frac{w \cdot h}{4}) \quad (8)$$

La soluzione a questa problematica la si ottiene tramite un approccio leggermente diverso. Viene utilizzata una funzione $r(x, y, seed)$ deterministica per generare un seed dedicato a ogni coordinata, per poi generare un solo numero dal generatore casuale inizializzato con il risultato di $r(x, y, seed.$ Questa miglioria consente un numero di computazioni pari a $\Theta(1)$.

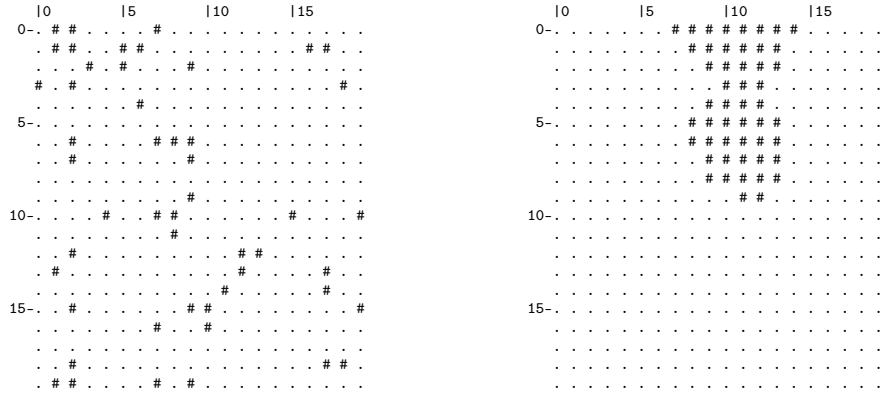
In particolare la funzione utilizzata è riportata all'algoritmo 1
[1, src/common/noise/perlin.rs:59]

Algorithm 1 Algoritmo di generazione del valore casuale per una coordinata data

```
1: function RANDOMXY(seed, x, y)
2:     derivedSeed ← HASH((seed, x, y))
3:     rng ← RNGFROM(derivedSeed)
4:     return NEXT32(rng)
5: end function
```

Questa funzione viene quindi utilizzata per generare un rumore di Perlin [3] tramite una serie di operazioni di scala e somma delle coordinate per ottenere diverse tipologie di conformazioni come mostrato dalla figura 4. I dettagli relativi alla generazione del rumore non rientrano nello scopo di questa relazione e di conseguenza non vengono trattati. Si può notare dalla

[1] Utilizzando una curva di hilbert per la computazione della sequenza date le sue proprietà di località.



(a) Limits (1529288299, 219) seed 4039822366 octaves 1 offset 0.1 size 1

(b) Limits (1390113848, 169) seed 42 octaves 1 offset 0.8 size 15

Figura 4: Esempi di configurazione del rumore con strutture degli ostacoli diverse

figura 4 come una configurazione con parametri diversi possa dare luogo a due strutture di ostacoli molto differenti: nella prima configurazione, grazie alla piccola dimensione del reticolo utilizzato per generare il rumore, si ottiene una disposizione molto distribuita, mentre nella seconda si osserva come, dato che la griglia viene generata con una sola cella di reticolo, essa presenta grande regolarità e compattezza.

La funzione descritta all'algoritmo 1 viene utilizzata per computare i vettori del reticolo necessari al calcolo del rumore di Perlin (figura 5). Successivamente, questo rumore viene quantizzato in modo da ottenere un solo valore per ogni cella della griglia (figura 6). A questo punto viene eseguito l'algoritmo 2 il quale, dato il generatore di rumore, la dimensione della griglia e il numero di ostacoli voluti, restituirà l'n-esimo minimo dei risultati della funzione rumore e le coordinate della cella nel quale questo viene calcolato [1, src/instance_gen/main.rs:22].

Il principio di funzionamento dell'algoritmo sopra definito è il seguente: viene mantenuto in memoria un max-heap sulla cui testa sarà presente l'n-esimo valore minore. Questa struttura contiene elementi interi.

L'algoritmo inizialmente inserisce i primi n valori nella struttura dati in quanto non vi è la necessità di effettuare comparazioni. A questo punto, se il numero di celle fosse meno del numero di ostacoli voluti, la lunghezza dell'heap dopo i cicli alle righe 3 e 4 sarà minore di n, da cui il ritorno di un fallimento. Contrariamente, dalla generazione della $(n + 1)$-esima cella,

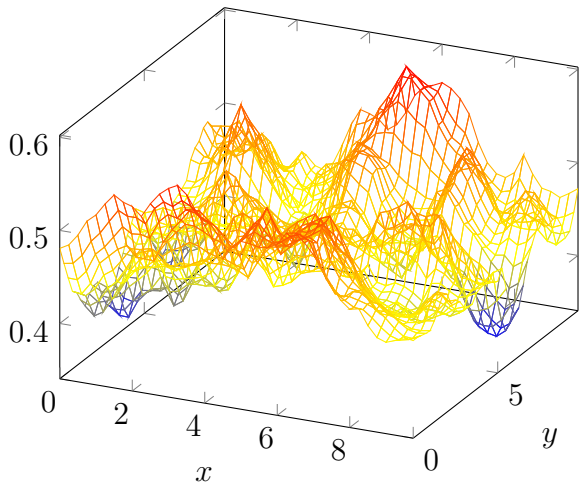


Figura 5: Plot del valore della funzione PerlinNoise nell'intervallo $[0, 10) \times [0, 10)$

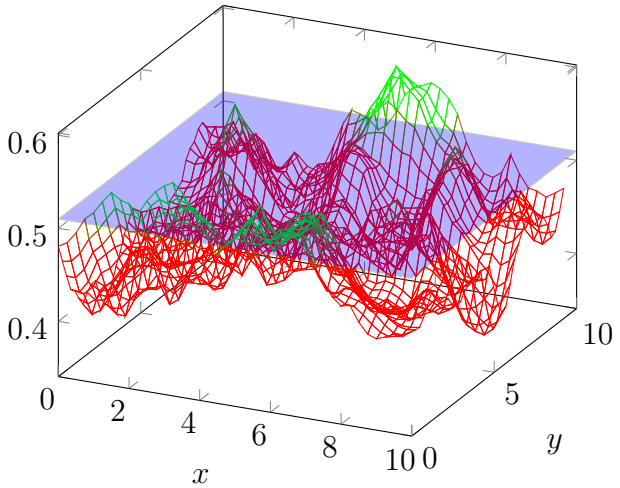


Figura 6: Plot del valore della funzione PerlinNoise quantizzata nell'intervallo $[0, 10) \times [0, 10)$

Algorithm 2 Algoritmo di generazione dei limiti descrittivi della griglia e degli ostacoli

```
1: function GenLimits(seed, w, h, n)
2:     maxHeap ←
3:     for j ∈ [0, h) do
4:         for i ∈ [0, w) do
5:             val ← randomXY(seed, i, j)
6:             if |maxHeap| < n then
7:                 heapPush(maxHeap, (val, (x, y)))
8:             else
9:                 if head(maxHeap)[0] > val then
10:                    heapPop()
11:                    heapPush(maxHeap, (val, (x, y))
12:                end if
13:            end if
14:        end for
15:    end for
16:    if len(maxHeap) < n then
17:        return failure
18:    end if
19:    return head(maxHeap)
20: end function
```

la testa dell'heap viene sostituita se il valore generato è minore dell'n-esimo minimo trovato fino alle iterazioni precedenti. Dopo la sostituzione, viene ricostruito l'heap (`heapify`) in modo da riottenere una struttura dati valida e coerente.

L'utilizzo di questo metodo, al momento della risoluzione dell'istanza, consentirà di determinare se una cella è un ostacolo in $\Theta(1)$ operazioni come osservabile dall'algoritmo 3 [1, src/common/field/field.rs:123].

Algorithm 3 Algoritmo di verifica presenza di un ostacolo alle coordinate date

```
1: function IsObstacle(x, y, limit, seed)
2:     return randomXY(seed, x, y) ≤ limit
3: end function
```

Ciò avviene con un costo di un utilizzo di memoria per il salvataggio degli ostacoli durante la generazione pari a $O(n)$, lineare nel numero di elementi, mentre un utilizzo di memoria $\Theta(1)$ durante le operazioni di risoluzione, in quanto risulta necessario salvare il valore limite e la prima cella dove questo valore accade per poter generare tutti gli n ostacoli.

Per quanto riguarda la complessità temporale, l'algoritmo di generazione deve valutare $\Theta(w \cdot h)$ valori, il che comporta un andamento quadratico nella larghezza/altezza della griglia (in caso di griglia quadrata), o lineare nel numero di celle.

È necessario sottolineare che questa ottimizzazione, sebbene teoricamente possa rappresentare ogni possibile conformazione della griglia dato il numero infinito di configurazioni iniziali possibili, non è computazionalmente ragionevole. Considerando i limiti effettivi imposti dalla dimensione dei dati utilizzati per il salvataggio delle istanze generate con il metodo discusso, è possibile calcolare il numero massimo di configurazioni rappresentabili. Tale valore deriva dal fatto che ogni possibile dimensione di griglia possa essere generata con 2^{64} possibili configurazioni iniziali e che siano rappresentabili griglie dalle dimensioni di 1×1 fino $2^{64} \cdot 2^{64}$. Tutto ciò senza considerare variazioni nei parametri di generazione del rumore. Le griglie più piccole di conseguenza riceveranno più configurazioni duplicate mentre le griglie più grandi potrebbero avere configurazioni non rappresentabili.

Riassumendo, è probabile che per una dimensione della griglia plausibile siano generabili tutte le sue configurazioni ma questa ricerca risulta essere troppo dispendiosa. Una soluzione a questo problema viene espressa alla sezione 2.2.

Non unicità della soluzione

Seppur non vi sia alcuna sicurezza sull'unicità dei valori generati dalla funzione $randomXY$, la probabilità che si verifichi una collisione di due valori casuali è data dalla somma tra la probabilità di collisione dell'hash utilizzato e dalla probabilità che il generatore di numeri pseudocasuali generi lo stesso valore a partire da due seed diversi. Tali probabilità sono strettamente legate dalla dimensione dell'output. In particolare, dato che il seed ha una dimensione di 64 bit [1][src/common/noise /perlin.rs:60], ne consegue che, per il paradosso del compleanno, vi sia una probabilità pari a circa $\frac{1}{2^{63}}$ che avvenga una collisione[2].

In pratica, questa limitazione garantisce con buona probabilità di poter generare una griglia di $2^{62} \times 2^{62}$ celle[3] senza che vi sia una singola collisione. È necessario tenere conto che, oltre a quanto riportato, la collisione deve avvenire nei primi n valori per avere un effetto. Considerate le dimensioni delle istanze e la difficoltà del problema, non vi sono dubbi che la generazione degli ostacoli è da ritenersi sufficientemente attendibile.

In alternativa è sempre possibile generare un limite euristico con il metodo sopra riportato e successivamente validare ed eventualmente ridurre tale limite al prezzo di ulteriore tempo di computazione e un ulteriore $\Theta(n)$ spazio in memoria, come dimostrato dall'algoritmo 4.

Nell'algoritmo presentato viene fatto uso di una TreeMap, in quanto risulta più efficiente nell'ottenimento della chiave massima alla riga 20, dal momento che le chiavi sono strutturate secondo un albero binario. Inoltre, utilizzando questa variazione, è necessario tenere traccia dell'ultima cella valida (valore ottenuto grazie alla funzione $getNthCell$, la quale ripercorre la griglia e restituisce la n-esima cella dove il valore ottenuto è pari a key). Ciò permette di risolvere con successo tutti i casi in cui vi sono più celle con valore esattamente pari al limite ma solamente una parte di esse può entrare a far parte degli ostacoli. In tal caso viene data precedenza alla cella più vicina all'origine sulle ordinate e successivamente sulle ascisse.

Tale funzione non viene inserita in questo documento in quanto le sue operazioni sono banali.

Parametri

I parametri configurabili che determinano la posizione e la struttura degli ostacoli sono i seguenti [1, src/common/noise/perlin.rs:87]:

[2]Ipotizzando la generazione di un'istanza ogni millesimo di secondo, sarebbero necessari circa 3 milioni di secoli per trovare una collisione

[3]Approssimativamente la massa in kg del più grande buco nero conosciuto

Algorithm 4 Algoritmo di verifica e rilassamento dei limiti descrittivi della griglia e ostacoli

```
1: function RelaxLimit(heap, seed, w, h, n, limit)
2:     count ← 0
3:     appearences ← TreeMap((i => 0)∀i ∈ heap)
4:     for j ∈ [0, h) do
5:         for i ∈ [0, w) do
6:             val ← randomXY(seed, i, j)
7:             if val ≤ limit then
8:                 appearences[val] ← appearences[val] + 1
9:                 count ← count + 1
10:            end if
11:        end for
12:    end for
13:    key ← limit
14:    if count > n then
15:        while count > n do
16:            appearences[key] ← appearences[key] − 1
17:            count ← count − 1
18:            if appearences[key] = 0 then
19:                delete(appearences[key])
20:                key ← maxKey(appearences)
21:            end if
22:        end while
23:    end if
24:    return (key, getNthCell(seed, w, h, key, appearences[key]))
25: end function
```

- **Octaves**: Numero di livelli del rumore per ogni valutazione della funzione. Intero maggiore di zero
- **Amplitude**: Moltiplicatore iniziale del rumore, Decimale
- **Persistence**: Riduzione progressiva dell'influenza dei livelli: Decimale compreso tra 0 e 1.
- **Frequency**: Valore iniziale del moltiplicatore delle coordinate al fine di non avere pattern all'interno del rumore generato, Decimale.
- **Lacunarity**: Moltiplicatore della frequenza per ogni livello.
- **Cell Size**: Dimensione della cella del reticolato rispetto alle coordinate della griglia. Intero maggiore di 0.

Un valore del parametro `cell_size` più piccolo comporta una struttura più sparsa degli ostacoli, mentre un valore vicino alla dimensione della griglia comporta generalmente pochi agglomerati di ostacoli. Questo perchè, essendo il rumore generato un rumore di Perlin, se la griglia del reticolo venisse impostata a una grandezza tale da includere tutte le coordinate della griglia, tutte le celle dipenderebbero dall'interpolazione di soli quattro vettori [3].

2.1.2 Estrazione casuale delle locazioni

Il problema di estrarre una cella casualmente dalla griglia tale che essa non sia un ostacolo trova la seguente soluzione.

Si vorrebbe che la variabile casuale rappresentante la cella fosse uniformemente distribuita tra le sole celle libere e traversabili. Questa condizione complica leggermente il problema in quanto non è immediato distinguere in modo efficiente le celle utilizzabili.

Un semplice algoritmo in grado di generare una coppia di coordinate con probabilità uniforme di corrispondere a qualunque cella della griglia si basa sull'ottenimento di un intero casuale nell'intervallo $[0, 2^{32}]$, per poi calcolarne il modulo rispetto all'altezza e alla larghezza della griglia rispettivamente per le ascisse e per le ordinate. L'algoritmo descritto potrebbe però selezionare una cella ostacolo.

Dato l'algoritmo 3, il quale determina se una cella è un ostacolo in funzione della configurazione in tempo costante, una soluzione alla questione precedente prende forma dell'algoritmo 5

L'algoritmo 5 presenta un utilizzo di memoria costante ma presenta un tempo di esecuzione dipendente dal fattore di occupazione

$$\sigma = \frac{\#ostacoli}{w \cdot h}$$

Algorithm 5 Algoritmo di estrazione casuale di una cella

```
1: function PickRandomCell(seed, limit, w, h)
2:     cell ← (next32(rng) mod w, next32(rng) mod h)
3:     while IsObstacle(cell[0], cell[1], seed, limit) do
4:         cell ← (next32(rng) mod w, next32(rng) mod h)
5:     end while
6: end function
```

In particolare la probabilità di estrarre un ostacolo, mantenendo l'ipotesi di uniformità delle distribuzioni, risulta essere esattamente σ.

Ne consegue che la condizione alla riga 3 risulta vera con una probabilità σ ogni volta. Non è possibile dunque stabilire deterministicamente la quantità di iterazioni necessarie a portare a termine l'esecuzione, tuttavia è possibile trovare quante iterazioni saranno necessarie per terminare con probabilità p, come illustrato nell'equazione 9. Tale soluzione viene trovata imponendo che la probabilità che per $T_p(\sigma)$ iterazioni vengano estratti sempre ostacoli ($\sigma^{T_p(\sigma)}$) sia minore di $1-p$, dove p è la probabilità di terminare (da cui risulta che la probabilità di estrarre un ulteriore ostacolo è minore di $1-p$).

$$
\begin{aligned}
P_{obstacle}(iteration\ k) &= \prod_{i=1}^{k} \sigma = \sigma^k \\
P_{exit}(iteration\ k) &= \prod_{i=1}^{k} (1-\sigma) = (1-\sigma)^k
\end{aligned}
\tag{9}
$$

$$
\sigma^{T_p(\sigma)} < 1 - p \implies T_p(\sigma) = log_\sigma(1-p)
$$

Tale calcolo viene riportato per un range di coefficienti σ e di probabilità p nella tabella 1.

Una soluzione deterministica si ottiene modificando l'algoritmo facendo sì che, una volta generata casualmente la prima cella, se essa è occupata da un ostacolo si cerchi la prima cella libera tra quelle a seguire. Diversamente, si ritorna la cella estratta inizialmente. Tale soluzione, sebbene non sia più performante, garantisce di terminare al più dopo $w \cdot h$ operazioni se $\sigma < 1$, in quanto percorre sequenzialmente la griglia. Questa soluzione non garantisce l'assoluta uniformità della distribuzione delle scelte generate, dato che le celle successive a una serie di ostacoli risultano avere una maggiore probabilità di scelta.

Si prenda come esempio la disposizione mostrata nella figura 7.

$\sigma \rightarrow$ $p \downarrow$	0.1	0.3	0.5	0.7	0.9	0.95
0.5	1	1	1	2	7	14
0.7	1	1	2	4	12	24
0.85	1	2	3	6	19	37
0.9	2	2	4	7	22	45
0.95	2	3	5	9	29	59
0.99	2	4	7	13	44	90

Tabella 1: Numero di iterazioni perché l'algoritmo `PickRandomCell` termini con probabilità p data una griglia con coefficiente di occupazione σ

```
  |0        |5              |0                  |5
0-. . . . . . . . . . .    0-1 1 1 1 1 1 1 1 1 1
  . . . . . . . . . . .      1 1 1 1 1 1 1 1 1 1
  # # # . . . . . . . #      0 0 0 3 1 1 1 1 1 0
  # # # # # . . . . . #      0 0 0 0 0 6 1 1 1 0
  # # # # . . . . . . .      0 0 0 0 5 1 1 1 1 1
5-# # # # . . . . . . .    5-0 0 0 0 4 1 1 1 1 1
  # # # # . . . . . . .      0 0 0 0 4 1 1 1 1 1
  # # # # . . . . . . .      0 0 0 0 4 1 1 1 1 1
  # # # . . . . . . . .      0 0 0 3 1 1 1 1 1 1
  # . . . . . . . . . .      0 2 1 1 1 1 1 1 1 1
```

(a) (b)

Figura 7: La figura mostra la disposizione della griglia (a) e la probabilità che ogni cella venga scelta (b) dall'algoritmo migliorato di scelta casuale `PickRandomCell` (Valori in percentuale)

2.1.3 Definizione dei percorsi degli agenti

La generazione dei percorsi degli agenti avviene scegliendo una mossa ciclicamente per ogni agente. Sia n il numero totale di agenti voluti e sia α_i la probabilità che, a ogni mossa, l'agente i-esimo si fermi e stia stazionario fino all'istante di tempo t_{max}.

La scelta iniziale della locazione dell'agente avviene mediante la versione migliorata sopra descritta dell'algoritmo. Durante la fase di sviluppo è stata ipotizzata la scelta di generare tali posizioni di partenza continuando la generazione della griglia con l'algoritmo `GenLimits`, dove i valori minori dell n-esimo minimo verrebbero considerati ostacoli e i valori compresi tra l'n-esimo e l'$n + a$-esimo minimo celle di partenza degli ostacoli, dove n corrisponde al numero di ostacoli e a corrisponde al numero di agenti. Tale soluzione garantirebbe un tempo di esecuzione pari a quello dell'algoritmo2, il quale corrisponde a $\Theta(w \cdot h)$ e un'occupazione di memoria di $\Theta(n + a)$. Il motivo per cui questa opzione non è stata preferita alla selezione casuale delle celle consta nella strutturazione del rumore. Tale strutturazione comporterebbe un insieme di celle di partenza per gli agenti la cui distribuzione non potrà mai essere uniforme.

Il processo di generazione risulta essere quindi svolto dall'algoritmo 6 [1, src/instance_gen/main.rs:45]

Si noti che viene mantenuto un HashSet delle posizioni finali degli agenti in modo da favorire la ricerca dell'esistenza di esse in tempo costante. All'interno dell'algoritmo 6 viene utilizzata una variante dell'algoritmo 5 la quale, oltre a includere l'ottimizzazione del paragrafo precedente, controlla alla riga 3 che la posizione estratta non esista nel set delle posizioni degli agenti la quale viene passata come ultimo argomento.

Successivamente, per ogni istante di tempo e per ogni agente, viene estratta la mossa successiva dell'agente i-esimo e successivamente aggiornata, rimuovendo la precedente dal set e inserendo la mossa appena estratta.

La generazione della mossa successiva, non mostrata in questo elaborato, sceglie casualmente con uguale probabilità una tra tutte le celle vicine alla posizione attuale dell'agente dove queste non sono ostacoli e non sono presenti all'interno dell'array *positions*.

A ogni mossa, con probabilità configurabile, un agente può fermarsi e rimanere stazionario fino alla fine del tempo t_{max}.

Grazie alle modalità di scelta delle mosse degli agenti, è matematicamente impossibile che essi compiano mosse invalide. Di fatto un solo agente viene mosso per volta, consentendo di evitare tutti quei casi dove due agenti devono scambiarsi di cella.

La memoria utilizzata dall'algoritmo 6 è calcolabile come segue:

Algorithm 6 Algoritmo di generazione percorsi degli agenti

```
1: function GenAgents(w, h, a, t_max, seed, limit)
2:     positions ← {} ▷ HashSet
3:     agents ← []
4:     len(positions) ← a
5:     len(agents) ← a
6:     for i ∈ [1, a] do
7:         pos ← PickRandomCell(seed, limit, w, h, positions)
8:         insert(positions, pos)
9:         agents[i] = Agent(pos)
10:    end for
11:    for t ∈ [1, t_max] do
12:        for i ∈ [1, a] do
13:            lastAgentPosition ← lastPos(agents[i])
14:            remove(positions, lastAgentPosition)
15:            newPos ← nextMove(agents[i], seed, limit, positions)
16:            insert(positions, newPos)
17:        end for
18:    end for
19:    return agents
20:
```

- Costo di salvataggio del vettore delle ultime posizioni: $\Theta(a)$
- Costo di salvataggio di ogni agente (a agenti):
  - Costo di salvataggio di ogni mossa: $O(t_{max})$

Risulta quindi un'occupazione di memoria pari a $O(a(t_{max}+1)) \approx O(a \cdot t_{max})$. L'utilizzo di $O(\cdot)$ invece che $\Theta(\cdot)$ è dato dalla possibilità per un agente di fermarsi. Questa condizione comporta che

$$S(a, t_{max}) = O(a(t_{max}+1))$$

e

$$S(a, t_{max}) = \Omega(2a) \approx \Omega(a)$$

in quanto ogni agente può avere come minimo una sola posizione associata.

Per il calcolo del tempo di esecuzione, utilizzando come parametri a e t_{max} e il coefficiente di occupazione della griglia σ, è possibile osservare che il ciclo alla riga 6 impiegherà con probabilità p per ogni iterazione un numero di iterazioni pari a

$$O\big(\min\big\{log_\sigma(1-p), w \cdot h\big\}\big) + 2\Theta(1)$$

Il ciclo alla riga 12 richede $3\Theta(1)$ in quanto l'algoritmo di selezione della mossa successiva [1, src/common/agent/agent.rs:41] ha un solo ciclo di al più nove iterazioni le quali eseguibili in tempo costante.

Risulta quindi che la complessità temporale dell'algoritmo 6 sia pari a

$$O\big(a \cdot \min\big\{log_\sigma(1-p), w \cdot h\big\} + a \cdot t_{max}\big) \leq O(a(w \cdot h + t_{max})) \approx O(a \cdot t_{max})$$

2.1.4 Metodo alternativo

Relativamente alla sezione 1.2.3, viene richiesto il calcolo del percorso ottimo verso la cella obbiettivo a ogni iterazione. Tale percorso è statico e non varia al procedere dell'algoritmo risolutivo. Risulta quindi possibile computare precedentemente alla soluzione tutti i percorsi ottimi dalla cella obbiettivo verso ogni cella della griglia (non ostacolo) di lunghezza al più t_{max}. Così facendo è possibile ottenere una struttura dati la quale permette di mappare ogni nodo a un suo successore nel percorso fino a giungere alla cella obbiettivo.

Questo permetterà di calcolare iterativamente il percorso più breve al momento della soluzione, senza dover considerare il problema di shortest-path a ogni iterazione, al prezzo di un utilizzo maggiorato della memoria.

La struttura dati utilizzata per salvare le associazioni sopra descritte consiste in una HashMap le cui chiavi sono le coordinate della cella e i valori

sono le celle di destinazione della mossa. Il calcolo dei percorsi ottimali avviene mediante algoritmo di Dijkstra nella sua implementazione con coda con priorità [1, src/instance_gen:83].

Nell'algoritmo vengono valutate solamente le celle che non rappresentano un ostacolo e solamente i percorsi distanti al più t_{max} dalla cella di arrivo.

Al fine di calcolare la complessità computazionale, possiamo considerare il grafo indotto dalla griglia il quale presenterà sempre $w \cdot h$ nodi (non considerando gli ostacoli) e al più $4(wh)$ archi. Tale numero viene trovato sapendo che, data una griglia di dimensioni w in larghezza e h in altezza, il numero di archi orizzontali sarà $(w-1)h$, considerando h righe e le transizioni tra le celle della stessa riga le quali equivalgono al numero di celle della riga meno una, e gli archi verticali che saranno $(h-1)w$. Infine si considerano i collegamenti diagonali i quali sono, per entrambe le direzioni, $(w-1)(h-1)$. Non vengono considerati i cappi in quanto non saranno mai inclusi in un percorso ottimale.

Risulta quindi che, escludendo gli ostacoli, all'interno del grafo compariranno un numero massimo di archi pari a

$$(w-1)h + (h-1)w + 2(w-1)(h-1) \approx 4(w \cdot h)$$

L'algoritmo di Dijkstra secondo la teoria raggiunge la soluzione in $O(V + E)$ iterazioni, corrispondente a una soluzione nel caso in esame pari a

$$O(5(\bar{w} \cdot \bar{h})$$

dove

$$\bar{w} = \min\left\{w, t_{max}\right\}$$

e

$$\bar{h} = \min\left\{h, t_{max}\right\}$$

Il risultato dell'elaborazione viene serializzato in un file compresso `*.aux` e messo a disposizione dell'algoritmo risolutivo al fine di velocizzare la soluzione, anche se così facendo vengono innalzati i costi di memoria di tale algoritmo, il quale dovrà caricare in memoria l'intera mappa generata per potervici iterare.

Durante lo sviluppo è stata ipotizzata la possibilità di accedere ripetutamente al file ausiliario la cui conseguenza sarebbe stata di non occupare memoria RAM ma sarebbe stata poco performante dati i ripetuti accessi al disco da effettuare. Inoltre si sarebbe reso necessario progettare un formato di serializzazione adatto all'accesso casuale e tale soluzione non è obbiettivo di questo corso.

2.2 Algoritmo risolutivo

In questa sezione vengono illustrarti gli algoritmi utilizzati al fine di risolvere le istanze generate alla sezione 2.1 con i cambiamenti necessari al fine di rendere tale algoritmo efficiente in termini di spazio allocato in memoria [4, p. 26]

Il processo risolutivo dell'istanza inizia con il caricamento delle configurazioni prodotte dal generatore di istanze le quali sono utilizzate per istanziare la griglia in funzione dei parametri del rumore e delle dimensioni.

All'interno dell'eseguibile di risoluzione è possibile specificare una diversa tipologia di griglia, denominata `CustomField` [1, src/common/field/field.rs:158], la quale permette di risolvere istanze non generate a partire da un rumore, bensì create a fini di test. Questa soluzione elimina le problematiche sorte alla sezione 2.1.1 relative alla rappresentazione di una qualsiasi configurazione della griglia. Il principio di funzionamento di questa soluzione consiste nell'organizzare un HashSet il quale contenga le locazioni degli ostacoli. Tale soluzione impiegherà $\Theta(1)$ operazioni per l'inserimento e la verifica (riportata nell'algoritmo 3) ma richiederà uno spazio in memoria di $\Theta(n)$, dove n rappresenta il numero degli ostacoli.

Dal punto di vista implementativo tale soluzione non comporta impieghi di tempo dedicati alla riallocazione in quanto la struttura dati avrà dimensione nota a priori della soluzione dell'istanza [1, src/common/field/field.rs:171].

Dopo il caricamento della struttura dati rappresentante la griglia, vengono caricati i percorsi degli agenti mediante una semplice lista di vettori, anch'essi di dimensione nota e di conseguenza senza problemi di preallocazione e riallocazione [1, src/solver/main.rs:244].

Successivamente il file contenente i percorsi precalcolati relativi alla sezione 1.2.3 viene eventualmente caricato in memoria.

Infine viene avviato il processo di risoluzione dell'istanza e il suo risultato viene inviato allo stream di output per poter eventualmente essere salvato su un supporto permanente come mostrato in figura 21.

2.2.1 Processo di risoluzione

È stato implementato l'algoritmo `ReachGoal` sulla base di quanto reso disponibile prima dell'inizio dello sviluppo e presente nell'appendice A. Tale elaborazione è riportata all'algoritmo 7 [1, src/solver/main.rs:132]

È possibile osservare che l'algoritmo presentato differisce da quello messo a disposizione nel testo dell'elaborato a fronte di una riscrittura atta a rendere il codice più leggibile e comprensibile.

Algorithm 7 Algoritmo risolutivo

```
1: function ReachGoal(field, agents, init, goal, t_max, aux)
2:     open ← [] ▷ MinHeap
3:     closed ← {} ▷ HashSet
4:     nodes ← {} ▷ HashMap
5:     nodes[init] ← VisitedNode(0, 0, nil)
6:     INSERT(open, (h(init), init, 0))
7:     while len(open) > 0 do
8:         (heuristic, current, t) ← POP(open)
9:         node ← nodes[current]
10:        ADD((current, t))
11:        if node = goal then
12:            break
13:        end if
14:        if t ≥ t_max then
15:            continue
16:        end if
17:        sol ← getAuxSolution(nodes, agents, current, goal, t, aux)
18:        if sol ≠ nil then
19:            return sol
20:        end if
21:        for neighbor ∈ NEIGHBORS(field, current) do
22:            if ¬isTraversable(agents, current, neighbor, t) then
23:                continue
24:            end if
25:            dest ← nodes[neighbor] or new VisitedNode(∞, ∞, nil)
26:            w ← calcWeight(current, neighbor)
27:            if CONTAINS(closed, dest)∧WEIGHT(node, t) + w ≥ WEIGHT(dest, t + 1)
    then
28:                continue
29:            end if =0
```

30: **if** WEIGHT$(node, t) + w <$ WEIGHT$(dest, t + 1)$ **then**
31: $w \leftarrow$ WEIGHT$(node, t) + w$
32: SET$(dest, t + 1, w, node)$
33: $nodes[neighbor] \leftarrow dest$
34: **end if**
35: **if** $\{(x, y) \in field | \exists (h, (x', y'), t) \in open, x = x' \wedge y = y' \wedge (x, y) = neigbor\}$ **then**
36: PUSH$(open, (h(neighbor, goal), neighbor, t + 1))$
37: **end if**
38: **end for**
39: **end while**
40: $(path, w, waits) \leftarrow$ RECONSTRUCTPATH$(nodes, goal)$
41: VERIFYPATH$(path, 0, t_{max}, agents, goal)$
42: **return** $(path, w, waits)$
43: **end function**

È evidente anche la rimozione delle righe 4-9 dell'algoritmo originale in quanto la mancanza di una posizione nella struttura dati *nodes* viene interpretata nell'avere valori di peso e tempo posti a infinito.

Relativamente alle strutture dati utilizzate, il set *open* viene implementato mediante un min-heap di terne contenenti il valore dell'euristica, le coordinate della cella in questione e il tempo nel quale la cella è stata raggiunta. Le comparazioni interne all'heap avvengono solamente in funzione dell'euristica e a parità di tale in funzione del tempo.

Il set *closed* viene implementato tramite un HashSet in modo da permettere la verifica e l'inserimento in tempo costante. Questa struttura dati viene creata pre-allocando almeno $\max\{|init_x - goal_x|, |init_y - goal_y|\}$ locazioni di memoria in quanto tale valore corrisponde al numero minimo di celle da attraversare e di conseguenza da inserire nel set per giungere alla soluzione.

L'ultima struttura dati utilizzata è una HashMap di oggetti denominati `VisitedNode` [1, src/common/field/visited_node.rs:7]. La loro funzione è offrire un oggetto in grado di mantenere al suo interno le associazioni tra tempo di visita e la tupla composta da peso di raggiungimento e cella predecessore. Questi oggetti offrono all'algoritmo la possibilità di chiamare i metodi WEIGHT$(node, t)$ e PARENT$(node, t)$ i quali permettono di ritornare rispettivamente il peso e il predecessore al tempo t o, nel caso questi non esistano, un valore di default ∞ e **nil**.

L'associazione sopra descritta è stata inizialmente implementata per mezzo di una HashMap, successivamente questa struttura è stata cambiata in una

BTreeMap[4] al fine di permettere un'ottimizzazione della memoria utilizzata valutata nella sezione 3.2.1.

La chiamata VERIFYPATH(·) permette di asserire relativamente alla correttezza della forma del percorso (traversabilità, mosse illegali, nodi di partenza e arrivo), ma non relativamente alla sua ottimalità. Tale verifica viene presa in considerazione nella sezione successiva.

Modifica strutturale

Confrontando l'algoritmo sopra commentato con la sua controparte nell'appendice, è possibile osservare una lieve differenza alla riga 27 rispetto alla riga 25. Viene cambiata la condizione del costrutto `if` facendo sì che un nodo venga rielaborato anche se precedentemente inserito nel set dei nodi già espansi. Questa condizione permette di risolvere correttamente una vasta famiglia di istanze al costo di relativamente poche computazioni aggiuntive ($O(\#agenti)$).

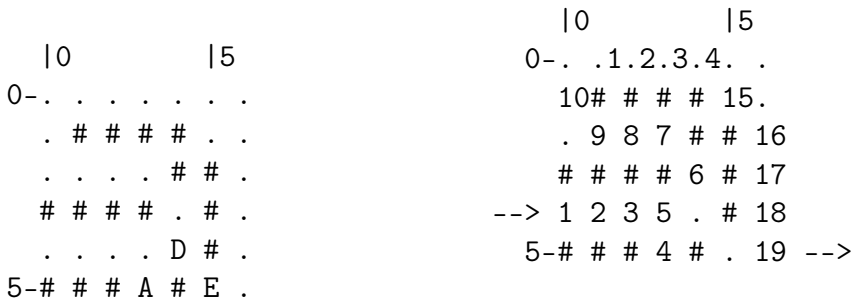
Un esempio di tale condizione viene mostrato nelle figure 8 e 9. Supponendo che l'euristica utilizzata sia la distanza di Euclide alla seconda, le celle espanse inizialmente saranno quelle di coordinate (0, 4), (1, 4), (2, 4) durante le prime tre iterazioni.

Alla quarta iterazione l'agente effettua la sua mossa portandosi nella cella (3, 4), la quale nell'iterazione precedente non era stata aggiunta al set *open* in quanto all'iterazione successiva la cella non sarebbe stata traversabile.

Questo comporta che la successiva cella espansa sarà quella di coordinate (3, 5) alla iterazione 3. A questo la cella in testa all'heap *open* sarà proprio la (3, 5) al tempo 3 in quanto è la cella più vicina alla cella obbiettivo. Questa iterazione comporta la scoperta della cella (3, 4), ora libera, provenendo dalla (3, 5). Alla quinta iterazione verrà espansa la (3, 4) per andare nella cella (4, 3) dopo di che il percorso possibile è solamente uno. Si giunge ora in un punto del percorso dove l'algoritmo preferirà elaborare mosse wait invece che proseguire essendo che le nuove celle continuano a distanziarsi sempre più dalla cella goal fino alla cella (0, 1). Una di queste rielaborazioni sarà la cella (2, 4) al tempo 3. Questa rielaborazione non porterà alcun risultato perché all'iterazione (3, 5) -> (3, 4) al tempo 3 la coppia ((3, 5), 3) è stata inserita nel set *closed* e per cui viene saltata seppur questa mossa porterebbe un beneficio e un peso minore al percorso trovato.

La soluzione per queste casistiche consiste nel reintrodurre il nodo già visitato nel set *open* se questa mossa portasse un miglioramento in termini di peso di visita del nodo stesso. Essendo che un percorso ottimo è collezione di

[4]Mappa associativa dove le chiavi vengono ordinate secondo un albero auto bilanciante [6]



(a) Percorso dell'agente

(b) Percorso ottimale trovato dall'algoritmo

Figura 8: Istanza di test #1, nessuna modifica all'algoritmo

sotto percorsi anch'essi ottimi, migliorando il peso di raggiungimento di un nodo del percorso assicuriamo che il percorso finale generato dall'algoritmo sia migliore.

Si noti che questa condizione ha effetto solamente se venisse scelta un'euristica non accurata. In caso contrario, utilizzando la distanza di Čebyšëv, l'algoritmo A* risulta essere ottimo e di conseguenza non si arriverà mai all'avverarsi della condizione addizionale imposta.

Questa addizione è stata fatta solamente per completezza dell'algoritmo rendendo così valide le soluzioni trovate con una qualsiasi euristica.

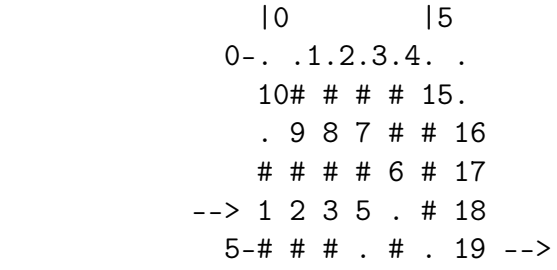


Figura 9: Istanza di test #1, modifica all'algoritmo alla riga 27, distanza euclidea

Soluzione non "greedy"

Sempre per completezza nel caso in cui l'algoritmo venisse invocato con un'euristica non accurata, è stata inserita una variabile di scelta per la risoluzione del problema in modo non "ingordo". Questa, unita alla modifica

descritta in precedenza, consente di esplorare ogni percorso rendendo A* effettivamente simile all'algoritmo di Dijkstra. Tale funzionalità non trova un risvolto di utilità ai fini del problema, ma consente di verificare la correttezza dell'algoritmo in fase di sviluppo permettendo di confrontare i risultati delle due esecuzioni (greedy e non greedy) consentendo così di validare i risultati con un discreto livello di confidenza.

2.2.2 ReconstructPath

L'algoritmo 8 ha il compito di ricostruire e ritornare il percorso ottimale a partire dalla struttura dati *nodes* generata dall'algoritmo 7. È bene notare che all'interno di tale struttura dati sono presenti oggetti i quali associano il tempo di visita di un nodo con il suo parente e peso di raggiungimento.

Questa strutturazione consente all'algoritmo, di seguito riportato, di ripercorrere il percorso ottimo a ritroso al fine di ricostruire la sequenza di celle.

Questa funzione non è responsabile della verifica della correttezza del percorso.

L'algoritmo iterativamente richiama la funzione PARENT$(\cdot)$ sul nodo in esame la quale ritorna il nodo parente o **nil** in caso di raggiungimento del nodo init[5].

A ogni iterazione l'elemento viene inserito in testa alla coda, la quale operazione, dato che la lunghezza della soluzione è nota in fase di creazione, equivale a inserire l'elemento all'indice $t-i$ del vettore, dove i indica il numero di iterazioni svolte, e successivamente viene aggiornato il nodo corrente con il suo predecessore prima di ricominciare il ciclo while.

I valori ritornati dall'algoritmo consistono nel percorso, nel peso del percorso e nel numero di mosse "wait".

Si noti che il tempo e il peso del percorso sono noti sin dalla riga 1 dell'algoritmo essendo l'unica associazione presente nella mappa della cella goal.

Si tratta di un algoritmo di complessità $\Theta(n)$ e occupazione di memoria $O(w \cdot h \cdot t_{max})$: la seconda è data dalla dimensione massima della struttura dati *nodes*, mentre la prima è data dal fatto che il percorso è univoco e già determinato nella struttura dati. Ne consegue che l'algoritmo 7 consista nella semplice iterazione su ogni elemento di tale percorso risultando quindi di complessità lineare.

[5] Algoritmo 7, riga 5

Algorithm 8 Algoritmo di ricostruzione percorso finale

```
1: function ReconstructPath(nodes, goal)
2:     node ← nodes[goal]
3:     t ← bestTime(node)
4:     w ← weight(node, t)
5:     queue ← []
6:     len(queue) ← node
7:     waits ← 0
8:     while true do
9:         current ← coordinates(node)
10:        pushFront(queue, current)
11:        t ← t − 1
12:        node ← nodes[parent(node, t)]
13:        if node == nil then
14:            break
15:        end if
16:        if current == coordinates(node, t) then
17:            waits ← waits + 1
18:        end if
19:    end while
20:    return (path, w, waits)
21: end function
```

2.2.3 Metodo alternativo

Per soddisfare il problema descritto nella sezione 1.2.3 e sulla base di quanto già discusso nella sezione 2.1.4, le righe 17-21 permettono di leggere i dati contenuti nel file ausiliario generato dal binario `instance_gen` (ref. fig. 21)

Sfruttando le funzionalità di serializzazione del linguaggio e delle sue librerie è possibile caricare in memoria la mappa che associa a ogni cella il suo successore nel percorso ottimo verso l'obbiettivo *goal*. Questo, come precedentemente accennato, è possibile dato che l'algoritmo di Dijkstra permette di calcolare il percorso ottimo a partire da una cella verso tutte le altre. Considerato che il grafo indotto dalla griglia non presenta archi orientati, ogni percorso ottimo da una cella c verso c' è anche ottimo dalla cella c' verso la cella c.

È quindi possibile scrivere la funzione 10 il cui compito è quello di valutare l'esistenza del percorso da una cella *current* verso *goal* e, in caso positivo, di ritornare la concatenazione del percorso trovato da *init* a *current* ricostruito tramite l'algoritmo 8 con il percorso trovato da *current* a *goal*.

Affinché sia possibile descrivere la funzione 10 risulta necessario poter verificare la presenza di un percorso attraverso la mappa ausiliaria. Il percorso così ottenuto sarà ottimo in senso assoluto in quanto calcolato senza la presenza degli agenti e, conseguentemente, senza mosse di attesa. Nel caso la cella *current* non fosse presente nella mappa ausiliaria è possibile ritornare fallimento immediatamente. La sua mancanza sta a significar che, nemmeno trascurando la presenza degli agenti, il percorso ottimo sarà più lungo di t_{max}.

Tale funzionalità di estrazione del percorso parziale è riportata successivamente all'algoritmo 9 [1, src/solver/main.rs:117].

Questo algoritmo è un semplice attraversamento del percorso e di conseguenza avrà complessità lineare nella lunghezza del percorso trovato.

Il percorso finale, dato dalla concatenazione del percorso identificato dalla struttura dati ausiliaria e da quello ricostruito nelle iterazioni precedenti dall'algoritmo 7, viene ritornato dall'algoritmo 10 [1, src/solver/main.rs:157]

È possibile notare come il ritorno dell'algoritmo 9 consista nell percorso dal successore di *current* fino a *goal*. Successivamente, se il percorso rilassato risultasse valido, condizione verificata dall'algoritmo 10, viene ritornata la concatenazione dei due percorsi, la somma dei loro pesi e il numero di mosse di attesa invariato, in quanto il percorso rilassato non ne può contenere.

Algorithm 9 Algoritmo di ottenimento di un percorso rilassato dalla struttura dati ausiliaria

```
function GetAuxPath(start, aux)
    path ← [] ▷ Stack
    next ← aux[start]
    w ← calcWeight(start, next)
    if next = nil then
        return failure
    end if
    while next ≠ nil do
        push(path, next)
        w ← w + calcWeight(next, aux[next])
        next ← aux[next]
    end while
    return (path, w) ▷ path as array
end function
```

Algorithm 10 Algoritmo di una soluzione valida a partire dal percorso rilassato

```
function getAuxSolution(nodes, agents, current, goal, t, aux)
    (relaxed, w) ← GetAuxPath(current, aux)
    if relaxed = failure then
        return nil
    end if
    if ¬verifyPath(relaxed, t, t_max, agents, goal) then
        return nil
    end if
    (partialPath, partialW, waits) ← reconstructPath
    return (partialPath||relaxed, partialW + w, waits)
end function
```

3 Analisi

Nella sezione seguente vengono mostrati e analizzati i risultati dell'esecuzione degli algoritmi sopra illustrati e discussi a fronte di istanze di test. Particolare enfasi è stata posta al consumo di memoria da parte dell'algoritmo.

L'obbiettivo ultimo dell'elaborato consiste nel limitare l'utilizzo di memoria dell'algoritmo `ReachGoal` a fronte dell'incremento delle dimensioni dell'istanza, numero di ostacoli e lunghezza della soluzione.

La dimensione della memoria occupata viene acquisita mediante memory-profilig eseguito con il tool "`heaptrack`".

3.1 Algoritmo di generazione delle istanze

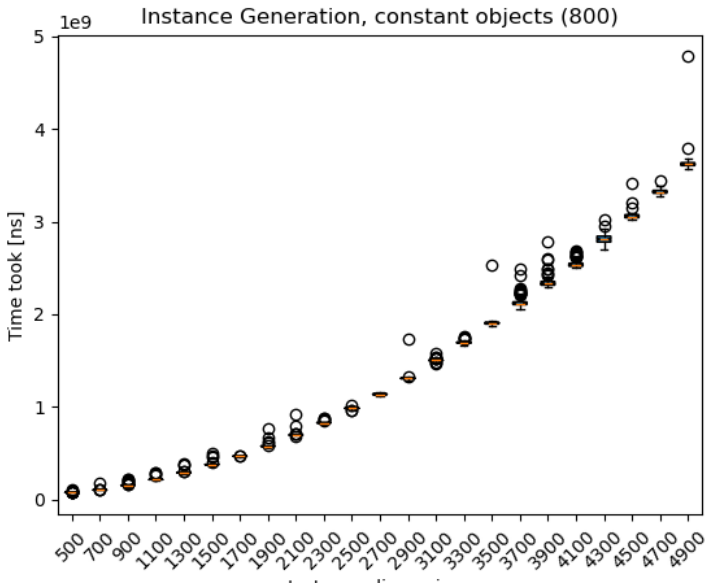
Al fine di valutare le performance temporali e spaziali dell'algoritmo di generazione vengono create 100 istanze casuali per ogni dimensione della griglia da 500x500 fino a 5000x5000 con incrementi di 100 al fine di garantire una buona base statistica. Ogni singola esecuzione viene quindi monitorata sia in termini di tempo di esecuzione che in termini di utilizzo di memoria.

In un primo test, atto a valutare l'utilizzo di memoria a fronte dell'incremento della dimensione dell'istanza e con numero di ostacoli costante (800 ostacoli) risulta che l'utilizzo sia constante come anticipato nella sezione 2.1.1. Nella figura 10b è prominente un salto dell'utilizzo di memoria, comunque contenuto, dovuto alle ottimizzazioni di allocazione delle strutture dati dinamiche e dall'allocazione delle strutture dati relative alla serializzazione dell'istanza generata.

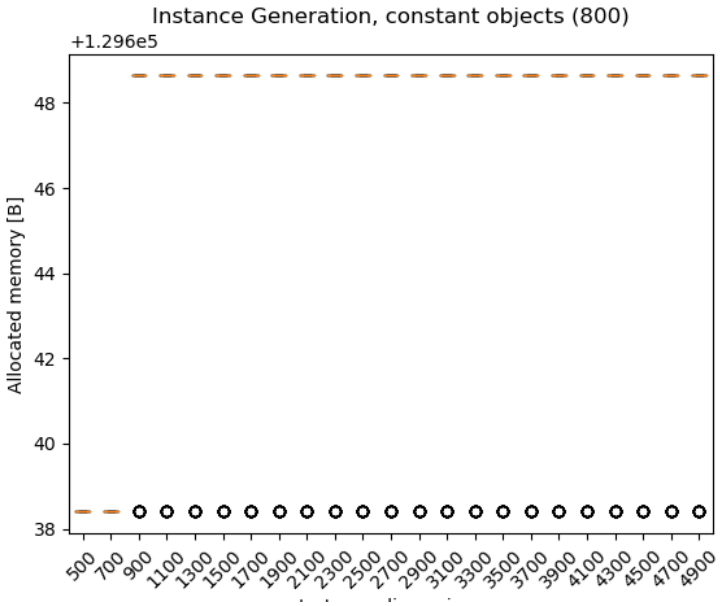
Relativamente al tempo di esecuzione del test (figura 10a) è evidente l'incremento quadratico dato dall'aumentare della dimensione della griglia

Secondariamente, al fine di dimostrare definitivamente la crescita lineare dell'utilizzo di memoria dell'algoritmo di generazione della griglia, esso viene eseguito incrementando in passi di 1000 il numero degli ostacoli e dimensionando la griglia di conseguenza (per un numero n di ostacoli la griglia avrà dimensioni $\lceil\sqrt{n}\rceil \times \lceil\sqrt{n}\rceil$).

Questo test, i cui risultati vengono riportati in figura 11 e è evidente il carattere lineare della memoria necessaria a conferma dei calcoli precedentemente riportati. Inizialmente l'utilizzo della memoria è mantenuto costante inizialmente dal fatto che l'utilizzo della stessa a fine di serializzazione e de serializzazione sovrasta l'utilizzo da parte dell'algoritmo. L'utilizzo della memoria da parte dell'algoritmo viene quindi nascosto essendo che vengono registrati solamente i picchi di utilizzo.



(a) Tempo impiegato in funzione della dimensione dell'istanza con numero di ostacoli costante



(b) Memoria allocata in funzione della dimensione dell'istanza con numero di ostacoli costante

Figura 10: Risultati relativi all'analisi dell'eseguibile instance_gen con numero di ostacoli costanti.

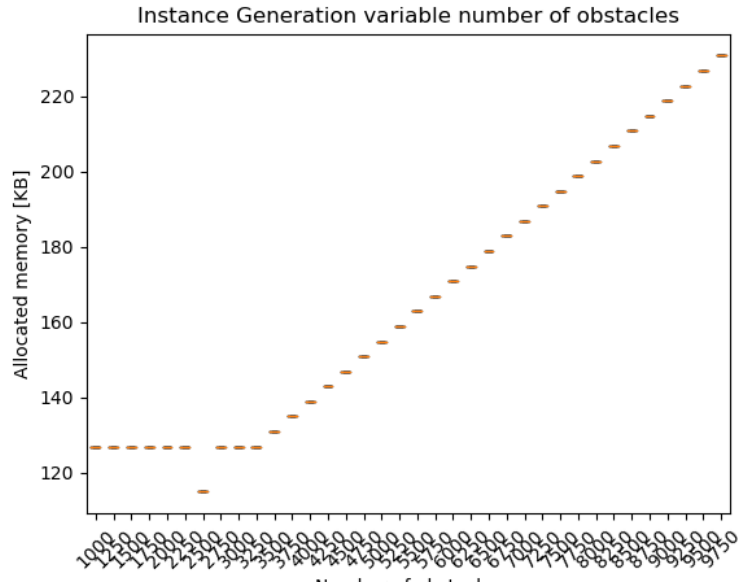


Figura 11: Risultati relativi all'analisi dell'eseguibile instance_gen con numero di ostacoli incrementale.

I parametri di configurazione del rumore per la prima classe di test sono stati mantenuti come specificati di default a eccezione della dimensione della cella, posta a un valore di 5, per mantenere una distribuzione uniforme degli ostacoli.

3.2 Algoritmo risolutivo

3.2.1 Scelta dell'euristica

Durante lo sviluppo sono stati messi alla prova tre diversi tipi di euristiche:

- Metrica Euclidea (elevata alla seconda)
- Distanza Diagonale
- Distanza di Čebyšëv.

Dell'elenco precedente solamente la seconda e terza distanza riportano in una soluzione ottimale, mentre la prima porta a condizioni di errore.

Per valutare le performance delle euristiche sopra descritte, sono state create delle istanze ad-hoc in grado di portare l'algoritmo all'estremo: La distanza Euclidea viene messa a dura prova da istanze formate come illustrato

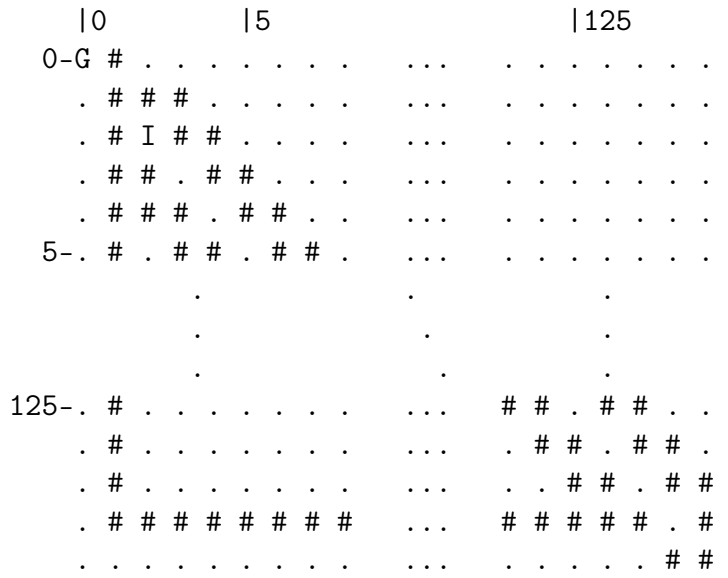


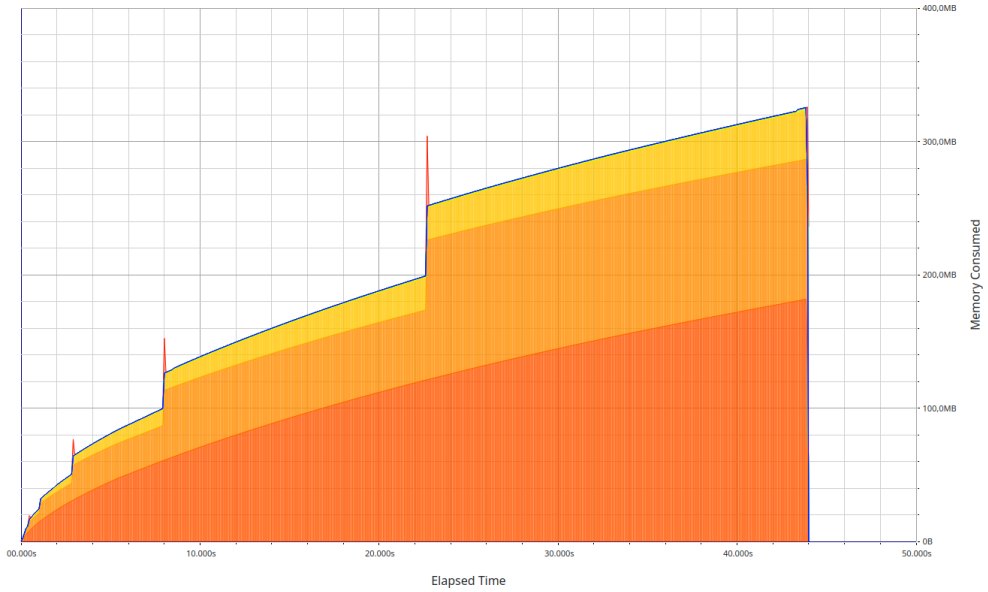
Figura 12: Costruzione dell'istanza per la valutazione dell'euristica euclidea. La cella I indica la cella iniziale e la cella G indica la destinazione

in figura 12. Questa istanza causa all'algoritmo risolutivo configurato per utilizzare l'euristica euclidea di rivalutare mosse "wait" perché più vicine alla cella di arrivo prima di procedere verso l'effettivo passo della soluzione.

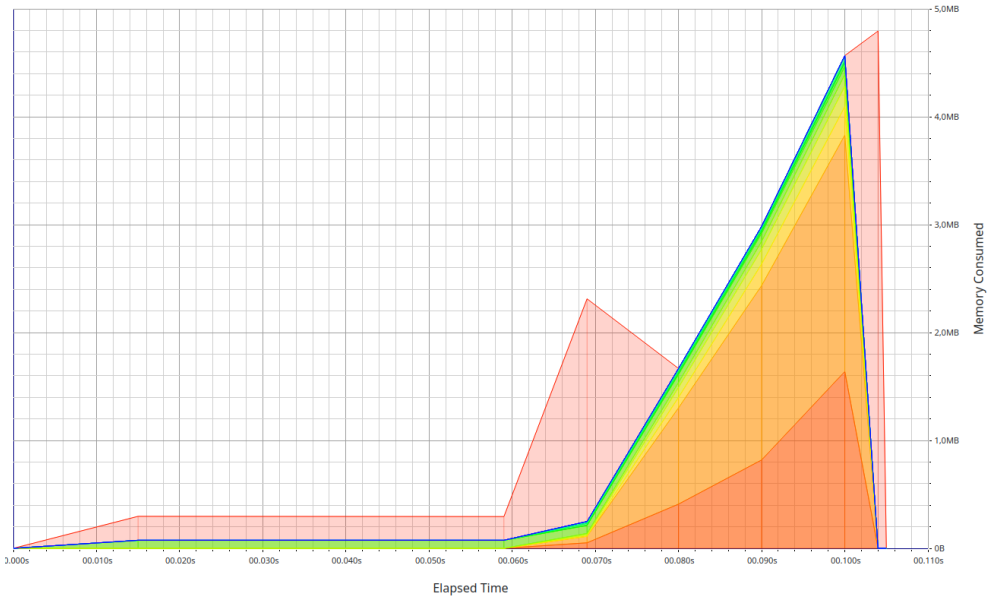
Per questo motivo è possibile osservare un utilizzo di memoria di 323MB per la risoluzione dell'istanza e un tempo di esecuzione di 45 secondi come riportato dalla figura 13a.

La stessa istanza risolta con l'euristica diagonale, i cui risultati sono consultabili alla figura 13b riporta figure di allocazione della memoria notevolmente ridotte. L'occupazione totale dell'esecuzione è stata 4.8MB e un tempo di risoluzione di 0.1s. Si noti inoltre che il picco di utilizzo di memoria è dato dalle strutture dati di serializzazione del risultato e non dall'esecuzione dell'algoritmo in sé.

L'euristica della distanza di Čebyšëv riporta risultati simili ma con tempi superiori essendo che questo metodo stima solamente con il massimo delle differenze delle coordinate. Questa scelta comporta l'espansione inefficiente in particolari casi. Per esempio le celle (10, 10) e (10, 0) sono a pari distanza secondo l'euristica in questione dalla cella (0, 0). In ogni caso, utilizzando questa euristica non vi sono grosse differenze nell'allocazione di memoria.



(a) Occupazione della memoria nel tempo con euristica euclidea



(b) Occupazione della memoria nel tempo con euristica diagonale

Iteration	Open Set Content (cell, time, heuristic)
0	((2, 2), 0, $2\sqrt{2}$)
1	((2, 2), 1, $1 + 2\sqrt{2}$), ((3, 3), 1, $3\sqrt{2}$)
2	((3, 3), 1, $3\sqrt{2}$), ((2, 2), 2, $2 + 2\sqrt{2}$), ((3, 3), 2, $1 + 3\sqrt{2}$)
3	((2, 2), 2, $2 + 2\sqrt{2}$), ((3, 3), 2, $1 + 3\sqrt{2}$), ((3, 3), 3, $2 + 3\sqrt{2}$), ...
...	...

Tabella 2: Evoluzione dell'heap *open*

Calcolo delle mosse "wait"

All'interno della consegna finale degli eseguibili è inclusa la possibilità di abilitare un'ottimizzazione dell'algoritmo risolutivo.

In particolare, tale ottimizzazione va a influire sulla gestione di mosse di attesa su nodi in istanti contigui.

Nella sua implementazione base, l'algoritmo `ReachGoal` salva il costo e l'istante temporale di provenienza in un nodo ogni qualvolta il peso di raggiungimento risulta essere migliorato. Questa condizione è sempre vera quando un nodo non è mai stato scoperto per un istante temporale dato anche nel caso di una mossa di attesa.

Tale comportamento risulta nel salvataggio di molteplici entrate nella BTreeMap [1, src/common/field/visited_node.rs:9] che associa il tempo di accesso al nodo visitato con il peso e il relativo predecessore durante i periodi di permanenza sullo stesso nodo.

Tale comportamento è esasperato da un'istanza generata secondo quanto descritto alla figura 14, dove l'agente a ogni passo si allontana di $\sqrt{2}$ dalla cella di arrivo.

Come mostrato dalla tabella 2, le prime due mosse espanse saranno lo spostamento diagonale (avente costo $\sqrt{2}$) e l'attesa nella cella iniziale. Successivamente si verifica una situazione tale per cui risulta più conveniente valutare mosse stazionarie invece che più avanti nel cammino, comportando un continuo aggiornamento dei nodi iniziali con visite provenienti da se stessi.

L'ottimizzazione trovata consiste, nel caso il nodo non fosse mai stato visitato, nel valutare la possibilità da parte dell'agente di permanere su tale nodo dall'ultimo tempo noto al tempo richiesto. Se tale possibilità esiste, viene ritornato il peso dell'ultima visita addizionato al peso della permanenza per l'intervallo di tempo mancante.

Questa soluzione permette di ridurre notevolmente il consumo di memoria, anche se aumentano i costi temporali in quanto devono essere rivalutate le posizioni di tutti gli agenti al fine di stabilire la possibilità di ottimizzazione.

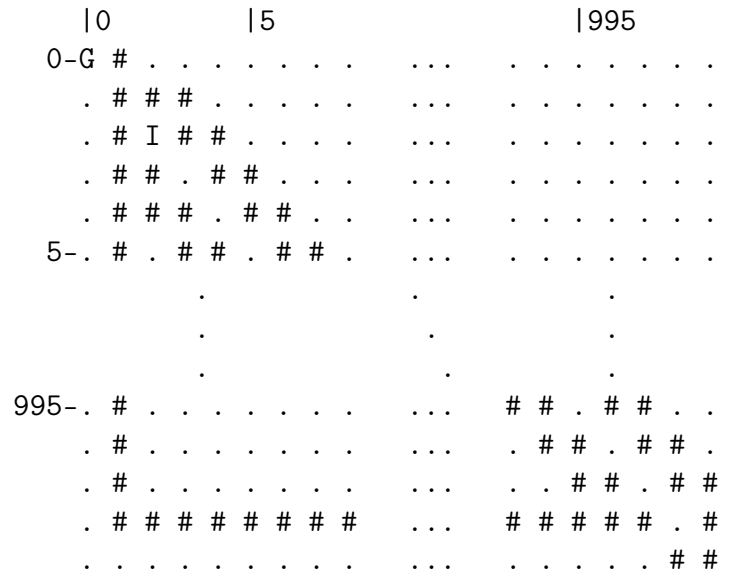


Figura 14: Costruzione dell'istanza per la valutazione dell'ottimizzazione di calcolo delle mosse wait. La cella I indica la cella iniziale e la cella G indica la destinazione

L'algoritmo di ottenimento del peso per il nodo v al tempo t viene riportato all'algoritmo 11 [1, src/common/field/visited_node.rs:58]

dove NEARESTKEYLE$(\cdot)$ rappresenta l'algoritmo di ottenimento della maggior chiave della mappa delle visite minore di un dato t. Tale algoritmo risulta essere efficiente grazie all'utilizzo di una TreeMap, la quale salva le proprie chiavi all'interno di un albero di ricerca.

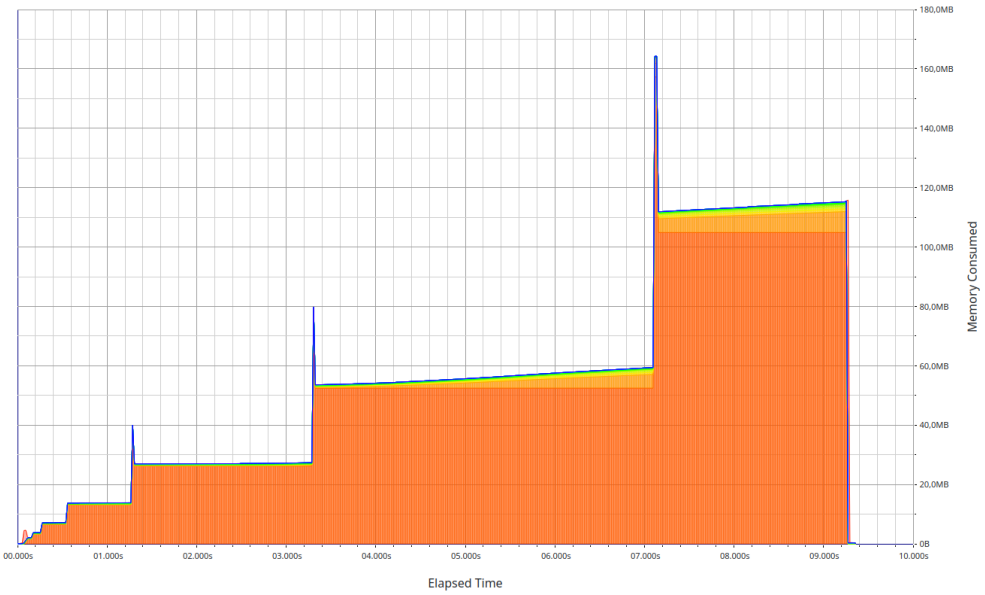
La funzione `canStay` verifica che nessun agente si ritrovi sulla cella in questione durante l'intervallo di tempo di permanenza.

I risultati di questa ottimizzazione sono disponibili alla figura 15. È possibile notare una diminuizione del picco di memoria utilizzata da 301MB a 164MB e un incremento del tempo di risoluzione di circa 4 secondi. Considerando l'obbiettivo di queste ottimizzazioni si può affermare che l'algoritmo ha subito un'ottimizzazione del 50% sul consumo di memoria.

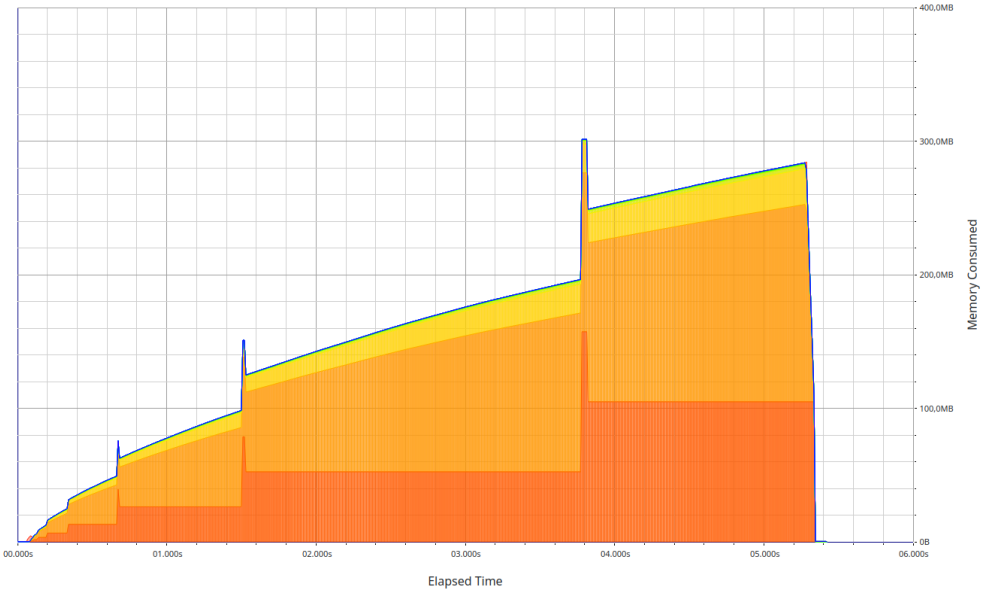
3.2.2 Metodo alternativo

Di seguito sono mostrati i test svolti sull'algoritmo risolutivo al fine di valutarne le prestazioni.

In un primo test si è voluto mostrare il miglioramento nel tempo di esecuzione e nel numero di stati aperti ed espansi derivato dall'utilizzo del me-



(a)



(b)

Figura 15: Risultati dell'elaborazione dell'istanza decritta alla figura 14 con (15a) e senza (15b) ottimizzazione

Algorithm 11 Algoritmo di calcolo del peso considerando la permanenza se possibile

```
 1: procedure GetWeightAtTime(v, t, agents)
 2:     lastVisitTime ← nearestKeyLE(v, t)
 3:     if lastVisitTime = nil then
 4:         return nil
 5:     end if
 6:     if lastVisitTime = t then
 7:         return v[t].weight
 8:         for i ∈ [lastVisitTime, t] do
 9:             if ¬canStay(agents, v.location, i) then
10:                 return nil
11:             end if
12:         end for
13:         return v[t].weight + calcWeight(v, v) · (t − lastVisitTime)
14:
```

todo alternativo, mentre in una seconda esecuzione si vuole mettere in luce il comportamento dei metodi risolutivi all'aumentare dell'affollamento della griglia.

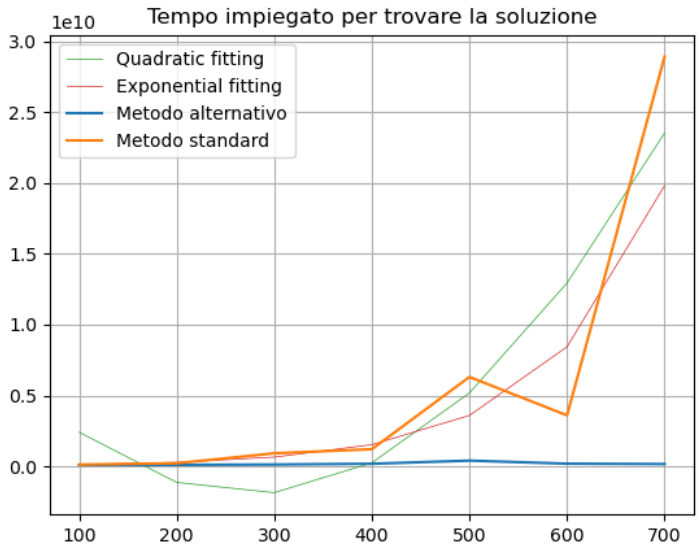
Al fine di produrre i dati presenti in figura 16 sono state risolte 100 istanze per ogni dimensione di griglia a partire da 100x100 fino a 1000x1000, incrementando di 100 la dimensione ogni volta. Il fattore di affollamento degli ostacoli σ rimane costante al valore 0.25, garantendo sempre il 75% della griglia disponibile.

Idealmente questo test dovrebbe essere svolto con un parametro σ_a pari a 0.1; ma, maggiormente per questioni legate alle tempistiche di esecuzione, è stata effettuata la scelta di limitare il numero di agenti massimi a 500. Questa scelta permette in ogni caso di valutare un trend nel confronto tra i metodi risolutivi.

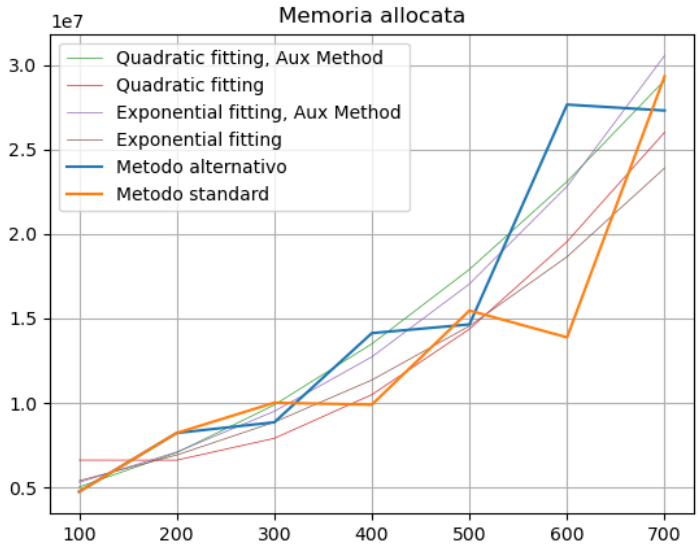
Dalla figura 16b è evidente come l'utilizzo del metodo alternativo comporti un utilizzo maggiore della memoria. Nel grafico vengono riportati come punti le medie dei risultati delle 100 iterazioni.

Dalla figura 16b, ulteriormente messo in evidenza dalla tabella 3, è osservabile un andamento quadratico di entrambi i metodi risolutivi nell'occupazione di memoria. Questo è dovuto dall'incremento del numero di celle nella griglia e conseguentemente del numero medio di celle visitate dall'algoritmo risolutivo.

Relativamente ai tempi di risoluzione, è osservabile dalla tabella 4 e dalla figura 16a un andamento esponenziale per l'algoritmo normale; mentre



(a) Tempo impiegato in funzione della dimensione dell'istanza con numero di ostacoli costante



(b) Memoria allocata in funzione della dimensione dell'istanza con numero di ostacoli costante

Figura 16: Risultati relativi all'analisi dell'eseguibile solver con numero di ostacoli costanti.

Metodo Alternativo

Andamento	a	b	c	R^2
$ax^2 + bx + c$	64.71	-19448.58	7916748.80	0.92
ae^{bx}	15.25	0.002		0.86

Metodo Standard

Andamento	a	b	c	R^2
$ax^2 + bx + c$	39.56	8454.26	3785808.74	0.90
ae^{bx}	15.19	0.003		0.85

Tabella 3: Risultati dell'interpolazione dell'andamento dell'occupazione di memoria

Metodo Standard

Andamento	a	b	c	R^2
$ax^2 + bx + c$	141293.63	-77849790.75	8778824252.10	0.79
ae^{bx}	17.73	0.008		0.90

Tabella 4: Risultati dell'interpolazione dell'andamento del tempo di esecuzione

non hanno rilevanza statistica i dati sul tempo risolutivo dell'algoritmo con metodo alternativo. È doveroso appuntare che l'ottimizzazione di cui alla sezione 3.2.1 è responsabile di un notevole incremento del tempo di risoluzione, anche se, dato il poco affollamento, è possibile affermare che tale parte del codice venga eseguita un numero di volte non determinante al fine dei risultati ottenuti.

Da queste esecuzioni è possibile dire però che il metodo alternativo in condizioni di basso affollamento consente una risoluzione in tempo quasi lineare dell'istanza.

Confrontando gli algoritmi risolutivi si riporta in media un utilizzo doppio della memoria utilizzando il metodo alternativo. Questo risultato conferma quanto ipotizzato alla sezione 2.1.4: la memoria occupata consiste interamente nella mappa di associazione precalcolata prima della soluzione, la quale deve essere caricata interamente in memoria affinché possa essere utilizzata.

Notevole miglioramento viene apportato invece nelle tempistiche di risoluzione. Come anche dimostrato dalla figura 17, il metodo alternativo si comporta molto bene tempisticamente in condizioni di basso affollamento della griglia trovando in meno iterazioni un percorso rilassato valido espandendo quindi pochi stati. La motivazione è data dal fatto che è altamente probabile che non vi sia alcuna interferenza da parte di agenti nel percorso

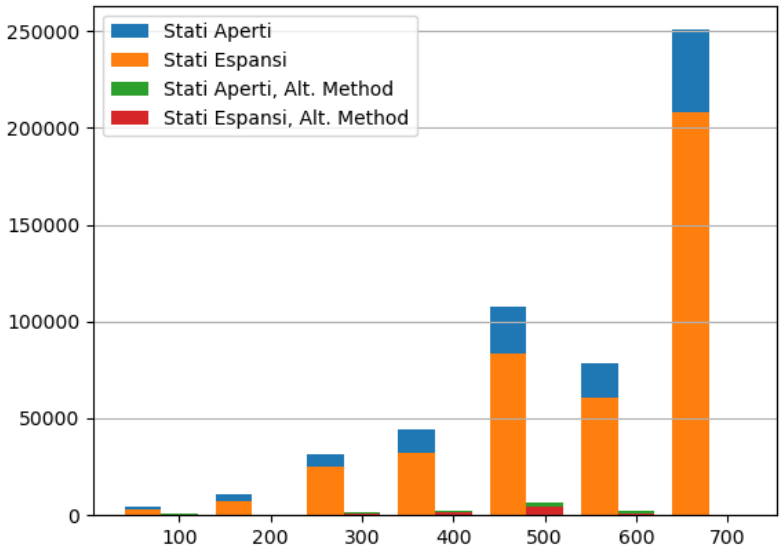


Figura 17: Comparazione del numero di stati aperti ed espansi in funzione della dimensione della griglia e metodo risolutivo

rilassato. È notevole la differenza di tempo impiegato nella risoluzione con e senza metodo alternativo.

Una seconda serie di test è stata svolta dove la dimensione della griglia è stata mantenuta costante variando invece i fattori di affollamento. È possibile osservare una costante perdita di efficacia del metodo alternativo.

Le istanze risolte alla figura 19 sono state generate ponendo $\sigma = 0.1$ e variando σ_a tra 0.1 e 0.95. Questo renderà costantemente più raro trovare un percorso diretto per il metodo alternativo, comportando quindi tempi di risoluzione e memoria simili o peggiori rispetto all'alternativa senza ottimizzazione.

Relativamente ai tempi di risoluzione si osserva un andamento lineare dovuto al fatto che queste istanze hanno mantenuto costante la dimensione, annullando gli effetti di incremento esponenziale delle metriche dovuti all'ingrandimento dell'istanza. È interessante osservare che il tempo di esecuzione dipende linearmente dal fattore di affollamento σ_a della griglia.

Sempre dalla figura 18a è possibile notare un lieve peggioramento delle prestazioni temporali dell'algoritmo con metodo alternativo a partire da un fattore σ_a pari a 0.3: questo è dovuto alle operazioni addizionali eseguite a ogni iterazione dell'algoritmo 7.

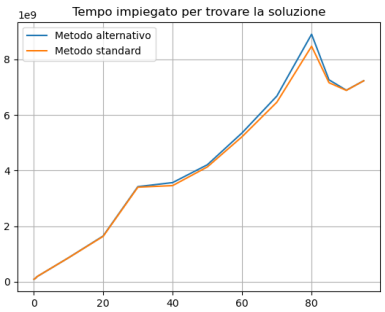
Non valutabile dal grafico si sottolinea un comportamento migliore per istanze poco affollate come riportato inizialmente in figura 16a. Questo effetto, dettagliato nella figura 18b è poco rimarcato date le dimensioni limitate delle istanze di test.

Relativamente all'occupazione di memoria si osserva un incremento lineare per le stesse motivazioni riportate precedentemente. L'incremento di utilizzo della memoria osservato precedentemente alla figura 16b non è evidente essendo che la memoria utilizzata al fine di risolvere il problema è di almeno due ordini di grandezza superiore alla dimensione della struttura dati ausiliaria.

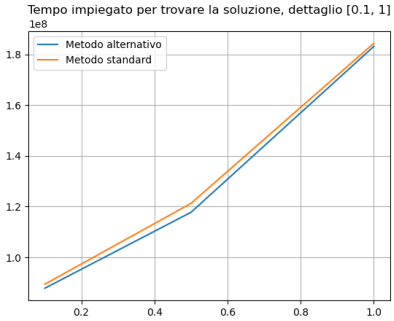
Osservando attentamente la figura 19 è possibile osservare alcuni comportamenti degni di nota e utili a confrontare le prestazioni del metodo alternativo con la sua controparte. Valutando la figura è evidente un andamento esponenziale del numero di stati aperti e/o espansi; comportamento atteso data la natura del problema. Successivamente è possibile osservare come con l'aumento di σ_a il rapporto tra il numero di stati aperti e numero di stati espansi tende all'unità. Questo comportamento, dettagliato in figura 20b, è classico della condizione di risoluzione dove vengono analizzati tutti i percorsi perché l'obbiettivo non è raggiungibile. Tale condizione comporta un tentativo di validare tutte le mosse per poi terminare con errore. Questa è una condizione attesa all'aumentare dell'affollamento in quanto è sempre meno probabile l'esistenza di una soluzione valida per l'istanza.

Infine, valutando la differenza tra i metodi risolutivi, emerge una tendenza a raggiungere lo stesso numero di stati espansi indipendentemente dall'algoritmo, effetto osservabile nel dettaglio alla figura 20a. La causa di questo comportamento è come detto precedentemente l'alta improbabilità di trovare una soluzione oltre che alla bassa probabilità di trovare un percorso rilassato senza agenti interferenti per l'aumento dell'affollamento.

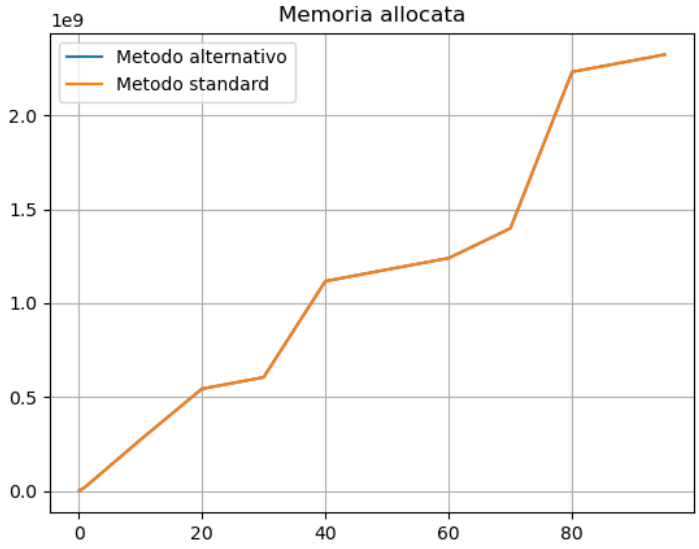
Risulta quindi che il metodo alternativo porta un miglioramento delle prestazioni dal momento in cui il fattore di affollamento della griglia è inferiore al 20-25%, per poi stabilizzarsi sulle stesse performance della versione senza questa ottimizzazione.



(a) Tempo impiegato al variare dell'affollamento



(b) Tempo impiegato al variare dell'affollamento, dettaglio



(c) Memoria allocata al variare dell'affollamento

Figura 18: Risultati relativi all'analisi dell'eseguibile solver al variare dell'affollamento.

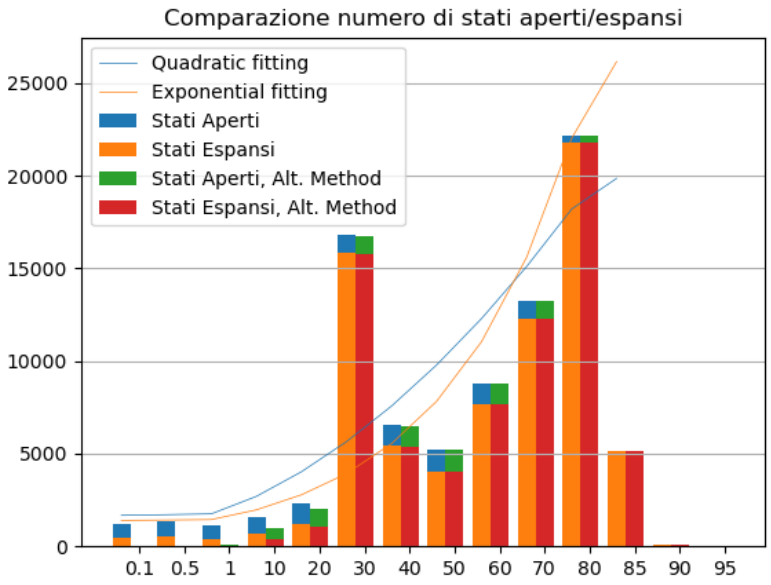
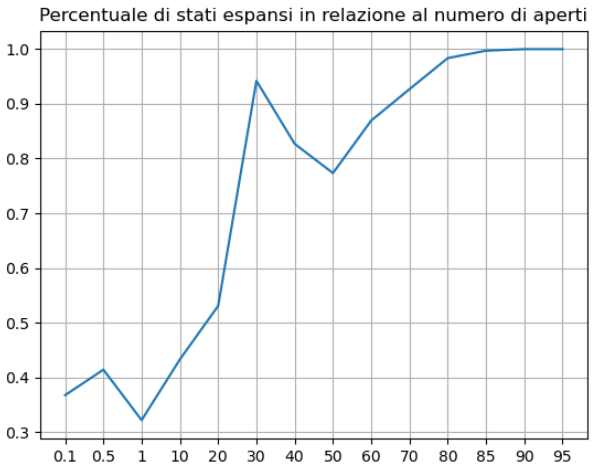
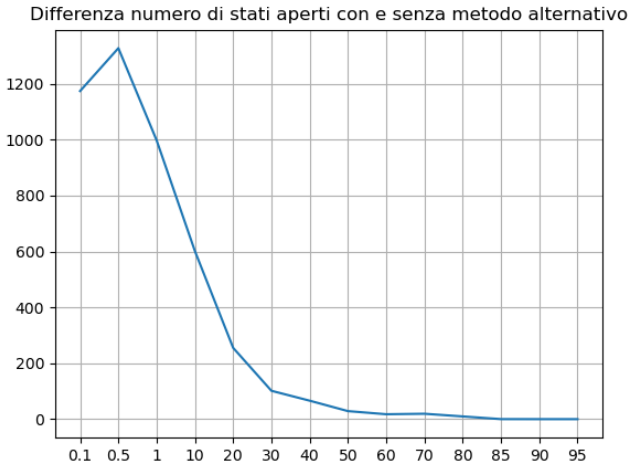


Figura 19: Comparazione del numero di stati aperti ed espansi al variare dell'affollamento



(a) Percentuale di stati espansi rispetto al numero di stati aperti.



(b) Differenza degli stati aperti al variare dell'affollamento per i due algoritmi.

Figura 20: Metriche derivate dalla figura 19

4 Conclusioni

A conclusione di questo elaborato si vuole riflettere sulle difficoltà riscontrate e l'ammontare di argomenti appresi.

Interfacciarsi a un nuovo linguaggio per la prima volta è stata una sfida non indifferente e ha richiesto un impegno notevole per continuare lo sviluppo a fronte di ostacoli e abbattimenti.

Questo approccio ha permesso in ogni caso di apprendere qualcosa in più rispetto a quanto previsto e, ora che questo lavoro è giunto al termine, è ragguardevole la soddisfazione personale che ne deriva.

Le principali difficoltà affrontate sono state relative ai paradigmi di programmazione derivati dalla struttura del linguaggio e la ricerca dell'ottimizzazione accompagnata dal continuo apprendimento delle funzionalità dell'ambiente messo a disposizione dal linguaggio Rust.

Non si nasconde che vi sono state alcune riscritture complete del codice cestinando completamente quanto ottenuto perché non ritenuto al livello necessario per essere presentato. Tuttora si vuole sottolineare come il codice possa subire un cospicuo numero di ottimizzazioni per migliorarne le performance.

In secondo luogo si è voluto presentare una serie di risultati statisticamente attendibili. Questo volere ha necessitato di un gran numero di ore di elaborazione per la risoluzione di tutte le istanze di test, senza considerare le volte dove per qualche errore quanto precedentemente prodotto risultava invalido o non utilizzabile.

Giunti al termine delle elaborazioni e della stesura di questa relazione, si spera che quanto contenuto sia adatto e venga validato positivamente.

Riferimenti bibliografici

[1] Stefano Fontana. *Elaborato*. URL: https://github.com/tetofonta/asd.

[2] Ingy döt Net Oren Ben-Kiki Clark Evans. *YAML Ain't Markup Language (YAML™) version 1.2*. YAML Language Development Team. URL: https://yaml.org/spec/1.2.2/.

[3] Ken Perlin. "An image synthesizer". In: *ACM Siggraph Computer Graphics* 19.3 (1985), pp. 287–296.

[4] Prof. Alessandro Saetti Prof. Marina Zanella. *Algoritmi e Strutture Dati, Testo dell'elaborato a.a. 2022-2023*. Università degli Studi di Brescia. URL: https://elearning.unibs.it/pluginfile.php/730327/mod_resource/content/4/Elaborato_2022_2023.pdf.

[5] Carol Nichols Steve Klabnik. "The Rust Programming Language". In: online, url: https://doc.rust-lang.org/1.71.1/book/title-page.html, 2023.

[6] Ronald L. Rivest Thomas H. Cormen Charles E. Leiserson. *Introduzione agli algoritmi.* Jackson Libri, 2003. ISBN: 88-256-1421-7.

List of Algorithms

Elenco delle tabelle

Elenco delle figure

A ReachGoal

Di seguito viene illustrata la prima implementazione dell'algoritmo "*ReachGoal*". Questo algoritmo sarà la base per lo sviluppo dell'implementazione finale la quale apporterà migliorie e ottimizzazioni descritte nella sezione 2.2.

```
1: procedure ReachGoal(G, w, Π, init, goal, t_max)
2:     Closed ← φ
3:     Open ← {(init, 0)}
4:     for t = 0 to t_max do
5:         for v ∈ V[G] do
6:             g(v, t) ← ∞
7:             P(v, t) ← nil
8:         end for
9:     end for
10:    g(init, 0) ← 0
11:    f(init, 0) ← h(init, goal)
12:    while Open ≠ φ do
13:        v, t ← Open[best(f(v', t'))], ∀v' ∈ V[G], t' ≤ t_max
14:        Open ← Open {(v, t)}
15:        Closed ← Closed ∪ {(v, t)}
16:        if v = goal then
17:            return ReconstructPath(init, goal, P, t)
18:        end if
19:        π_r ← relaxed(v, goal)
20:        if π_r is collision free then
21:            return ReconstructPath(init, goal, P, t)
22:        end if
23:        if t < t_max then
24:            for n ∈ Adj[v] do
25:                if (n, t + 10 ∉ Closed then
26:                    traversable ← true
27:                    for π ∈ Π do
28:                        if π_{t+1} = n ∨ (π_{t+1} = v ∧ π_t = n then
29:                            traversable ← false
30:                        end if
31:                    end for
32:                    if traversable = true then
33:                        if g(v, t) + w(v, n) < g(n, t + 1) then
```

```
34:                         P(n, t + 1) ← (v, t)
35:                         g(n, t + 1) ← g(nt) + w(v, n
36:                         f(n, t + 1) ← g(n, t + 1) + h(n, goal)
37:                     end if
38:                     if (n, t + 1) ∉ Open then
39:                         Open ← Open ∪ {(n, t + 1}
40:                     end if
41:                 end if
42:             end if
43:         end for
44:     end if
45: end while
46: return failure
47: end procedure=0
```

B Manuale d'uso

La sezione seguente sezione riguarda le procedure di installazione e utilizzo degli artefatti prodotti da questo progetto e annesse dipendenze.

B.1 Introduzione

Il progetto utilizza rust come linguaggio di sviluppo per le motivazioni esposte alla sezione 2. Lo sviluppo è stato interamente effettuato in ambiente Linux e non sono stati effettuati test di portabilità degli artefatti su altre piattaforme in quanto non richiesti.

Affinché sia possibile compilare ed eseguire i vari artefatti sono necessari i seguenti componenti:

- Una versione aggiornata del compilatore `rustc`. Durante lo sviluppo la versione utilizzata è stata `rustc 1.71.1 (eb26296b5 2023-08-03)`.
- Una versione aggiornata del gestore di pacchetti `cargo` (versione utilizzata `cargo 1.71.1 (7f1d04c00 2023-07-29)`).

Per eseguire gli script atti a produrre i risultati mostrati all'interno dell'elaborato è necessaria un'installazione di `python` aggiornata (versione utilizzata `3.11.3`) e annesso package manager `pip` (versione utilizzata `23.2.1`).

L'installazione degli strumenti non verrà trattata in quanto fortemente dipendente dalla piattaforma e dalla distribuzione, oltre che già sufficientemente trattata da svariate risorse in rete[6][7][8][9].

Si sottolinea che la maggior parte delle distribuzioni Linux supporta uno o più degli strumenti per l'installazione tramite il gestore di pacchetti di sistema della distribuzione stessa.

B.2 Compilazione

Il progetto consiste in due eseguibili compilabili separatamente tramite il gestore di pacchetti `cargo`.

La compilazione avviene con un singolo comando:

```
cargo build --profile release --package
      AlgorithmsAndDataStructures
```

[6] `https://www.rust-lang.org/tools/install`
[7] `https://doc.rust-lang.org/cargo/getting-started/installation.html`
[8] `https://www.python.org/downloads/`
[9] `https://pip.pypa.io/en/stable/installation/`

Alla compilazione sono disponibili diverse "features" utili per testare diversi comportamenti degli algoritmi con diverse implementazioni. Le funzionalità vengono descritte di seguito.

- Scelta dell'euristica. Solo una delle seguenti opzioni è consentita.
  - `diagonal_distance`: Se abilitata l'algoritmo risolutivo utilizzerà l'euristica della distanza diagonale.

$$\Delta x + \Delta y + (\sqrt{2} - 2) * \min\{\Delta x, \Delta y\}$$

  - `chebichev_distance`: Se abilitata l'algoritmo risolutivo utilizzerà l'euristica della distanza di Čebyšëv.

$$\max\{\Delta x, \Delta y\}$$

  - `euclidean_distance`: Se abilitata l'algoritmo risolutivo utilizzerà l'euristica della distanza euclidea alla seconda potenza.

$$(\Delta x)^2 + (\Delta y)^2$$

- `wait_move_weight_calc` Abilita l'ottimizzazione discussa alla sezione 3.2.1

Per selezionare una o più funzionalità al momento della compilazione è possibile aggiungere al comando di compilazione l'indicazione seguente

```
cargo build ...  --features <feature>[{,<feature>}*]
```

B.2.1 Architettura

Il processo generale di generazione e soluzione di un'istanza, mostrato in figura 21, consiste nella pre-generazione di tutti i parametri relativi all'istanza tramite il binario `instance_gen`. Esso prende in input i parametri definiti nella sezione **??** (come file o come parametri da linea di comando) e genera un file descrittore dell'istanza, descritto nella sezione **??** e opzionalmente un file ausiliario per velocizzare la soluzione dell'istanza pre-calcolando i percorsi migliori a partire dalla posizione di arrivo. Successivamente il binario `solve`, il cui funzionamento viene discusso a partire dalla sezione 2.2, sarà responsabile della soluzione effettiva dell'istanza e dell'output del file contenente il risultato.

B.2.2 Generazione dell'istanza

Un'istanza può essere generata a partire da un file descrittore della stessa o da un'invocazione dell'eseguibile `instance_gen` (disponibile nella sottodirectory `target/release/` dopo la compilazione).

Nel primo caso il file deve contenere uno o più documenti in formato YAML contenenti dati nel seguente formato:

```
[...]
---
id: string
kind: "settings"
seed: integer (opt)
obstacles: integer
time_max: integer
aux_path: path(string)  (opt)
size:
  width: integer
  height: integer
agents:
  number: integer
  stop_probability: integer (opt)
noise:
  octaves: integer (opt)
  persistence: float (opt)
  lacunarity: float (opt)
  amplitude: float (opt)
  frequency: float (opt)
  cell_size: integer (opt)
  offset: float (opt)
---
[...]
```

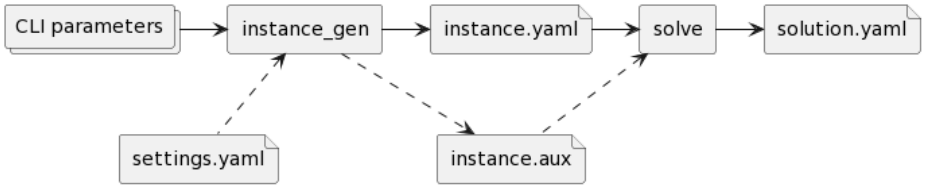


Figura 21: Diagramma illustrativo del processo di generazione e soluzione dell'istanza

Aternativamente è possibile invocare l'eseguibile passando parametri da linea di comando descritti di seguito.

- `-c,--config CONFIG` Nome del file di configurazione (formato YAML [2]) dal quale caricare le informazioni. Tale file sovrascrive completamente i parametri da linea di comando rendendoli inutilizzati.
- `-i,--config-id CONFIG_ID` identificativo della configurazione da caricare. Questo consente di contenere più configurazioni nello stesso file tramite l'utilizzo della funzionalità multi-documento del linguaggio yaml [2]. Se non specificato verrà caricata la prima configurazione trovata
- `-o,--aux-file AUX_FILE` Nome del file di output dei percorsi ausiliari precalcolati ref. 2.1.4
- `-e,--exhaustive` Se presente verranno valutati tutti i possibili percorsi invece che una ricerca "*greedy*"
- `-s,--seed SEED` Configurazione iniziale del generatore pseudocasuale
- `-w,--width WIDTH` Larghezza della griglia
- `--height HEIGHT` Altezza della griglia
- `-o,--obstacles OBSTACLES` Numero di ostacoli
- `-t,--tmax TMAX` Limite temporale massimo della soluzione
- `-a,--agents AGENTS` Numero di agenti
- `--agent-stop-probability AGENT_STOP_PROBABILITY` Probabilità che un agente si fermi e resti stazionario fino al limite temporale per ogni mossa
- `--octaves OCTAVES` ref. 2.1.1
- `--persistence PERSISTENCE` ref. 2.1.1
- `--lacunarity LACUNARITY` ref. 2.1.1
- `--amplitude AMPLITUDE` ref. 2.1.1
- `--frequency FREQUENCY` ref. 2.1.1
- `--cell_size CELL_SIZE` ref. 2.1.1

- `--offset OFFSET` ref. 2.1.1

Tale descrizione è ottenibile mediante il parametro `--help`.

È necessario osservare che ogni configurazione possiede un codice identificativo obbligatorio: questo risulta utile in fase di generazione dell'istanza permettendo di generare seguendo una data configurazione a partire da un file contenente multiple opzioni.

L'output di questa operazione consiste in un file il quale segue il seguente schema:

```
[...]
---
id: string
kind: "instance"
seed: integer
grid:
  width: integer
  height: integer
  obstacles: integer
  noise:
    octaves: integer
    persistence: float
    lacunarity: float
    amplitude: float
    frequency: float
    cell_size: integer
    val_limit: integer
    cell_limit: integer
    offset: float
aux_path: path(string) (opt)
agents:
  paths: (x, y)[][]
time_max: integer
init: (x, y)
goal: (x, y)
---
[...]
```

Nei file descrittivi dell'istanza vengono riportate tutte le configurazioni di generazione necessarie alla risoluzione con l'aggiunta dei limiti per l'identificazione degli ostacoli (sezione 2.1.1), celle di inizio e fine e percorsi degli agenti, rappresentati da una lista di posizioni per ogni agente.

L'output viene diviso su i due stream di ouput (stdout e stderr) di ogni applicazione consentendo di creare configurazioni complessa a piacere di piping e gestione degli stream.

Lo stream di errore viene utilizzato per comunicare informazioni di contorno ed errori all'utente, mentre lo stream di output verrà utilizzato per inviare il documento YAML rappresentante l'istanza. Se eseguito normalmente senza reindirizzazioni l'eseguibile stamperà a schermo il descrittore dell'istanza; è perciò consigliato l'utilizzo dell'operatore di redirect come mostrato nell'esempio seguente:

```
./target/release/instance_gen -c settings.yaml -i settingsId >
                        instance.yaml
```

B.2.3 Risoluzione

La risoluzione di un'istanza avviene richiamando l'eseguibile `solver`, sempre disponibile nella sottodirectory `target/release`, con i seguenti parametri:

- `-c,--config CONFIG` Nome del file di descrizione dell'istanza (formato YAML [2]) dal quale caricare le informazioni.
- `-i,--config-id CONFIG_ID` identificativo della configurazione da caricare. Questo consente di contenere più configurazioni nello stesso file tramite l'utilizzo della funzionalità multi-documento del linguaggio yaml [2]. Se non specificato verrà caricata la prima configurazione trovata

Come nel caso dell'eseguibile di generazione, l'output informativo verrà stampato sullo stream `stderr` mentre il risultato dell'elaborazione dell'istanza verrà reso disponibile sullo stream di output nel seguente formato in caso di successo:

```
[...]
---
kind: "solution"
expanded_states: integer
opened_states: integer
path_info:
  path: (x, y)[]
  weight: float
  time: integer
  waits: integer
---
[...]
```

Il significato dei parametri esportati è privo di ambiguità. Per ogni percorso vengono esportati il numero di stati espansi e aperti oltre che all'effettiva soluzione, il peso del percorso trovato, il tempo impiegato (numero di passi) e il numero di mosse di attesa nel percorso.

In caso di errore il formato di output è il seguente:

```
[...]
---
kind: "error"
expanded_states: integer
opened_states: integer
path_info: null
---
[...]
```

B.3 Test

Al fine di eseguire i test e generare i dati presentati in questo elaborato sono necessari ulteriori passi di preparazione:

1. Compilare gli eseguibili per la profilazione utilizzando l'indicatore `-profile profiling` nel comando di compilazione indicato nella sezione B.2.
2. Installare le librerie necessarie all'elaborazione con il comando `pip install -r requirements.txt`

Attenzione!

Il test eseguito dallo script `tests/solver_alt_method_sigma_incr.py` presenta una variabile di decisione relativamente alla cancellazione delle istanze generate dopo la loro risoluzione. Dato l'elevato numero di istanze generate e la loro dimensione, il mantenimento sul disco dei file relativi comporta un utilizzo di circa 150GB dello stesso. La quantità di ram utilizzata per l'esecuzione delle istanze "grandi", inoltre, è di circa 2GB. Si consigliano almeno 8GB di ram liberi per poter eseguire più soluzioni in contemporanea.

È possibile eseguire i test presenti nella directory `tests` mediante invocazione con l'interprete python dalla directory principale del codice.

```
find tests/ -name '*.py' | xargs -I {} python {}
```

Date le dimensioni e la quantità di dati utilizzata per avere risultati statisticamente accettabili non è possibile consegnare direttamente i file descrittivi delle istanze. Essendo che la generazione dipende da un seed principale, le

istanze possono essere rigenerate con risultati identici in un secondo istante a partire dagli script presenti nella directory di test.

C Struttura del progetto

Il codice consegnato è composto dai seguenti componenti:

- `Cargo.toml` File descrittivo del progetto per il gestore di pacchetti.
- `requirements.txt` File contenente le dipendenze alle librerie python per l'esecuzione dei test.
- `src/*` Codice sorgente
- `tests/*` Directory contenente gli script python di generazione dei grafici per l'analisi
  - `test_instance_gen.py` Verifica le performance dell'algoritmo di generazione in funzione della dimensione della griglia
  - `test_instance_gen_2.py` Verifica le performance dell'algoritmo di generazione in funzione dell'affollamento di ostacoli
  - `solver_alt_method_dim_increase.py` Verifica le performance e confronta i metodi risolutivi di istanze all'aumentare delle dimensioni
  - `solver_alt_method_sigma_incr.py` Verifica le performance e confronta i metodi risolutivi di istanze all'aumentare dell'affollamento degli agenti
- `settings.yaml` File di esempio per la generazione di istanze
- `custom_instances.yaml` File contenente una serie di casi particolari per il testing e la validazione dell'algoritmo.