Tesina esame “Algoritmi e strutture di dati”
Algoritmo per la generazione di labirinti

Carlo Tosoni
Giacomo Massini
Corso di ingegneria informatica,
Università degli studi di Perugia

Premessa

La tesina da noi proposta è una possibile soluzione al problema della generazione di labirinti, utilizzando un algoritmo che è stato trattato durante il corso di algoritmi e strutture di dati: l'algoritmo di Kruskal. Le celle del labirinto sono state in questo caso considerate come nodi di un grafo: se fra due nodi confinanti, u e v, non esiste nessun arco (u,v) che li colleghi, allora fra questi è presente un muro, in caso contrario è possibile spostarsi da u a v. L'algoritmo di Kruskal è un algoritmo goloso che viene utilizzato per individuare un minimum spanning tree (MST), cioè un albero che colleghi tutti i nodi del grafo in modo tale che la somma dei pesi degli archi percorsi sia minima. Nel calcolare un MST di un determinato grafo, l'algoritmo di Kruskal procede in questo modo:

1) si ordinano gli archi del grafo in ordine non decrescente di peso
2) viene selezionato il primo tra gli archi (ordinati) non ancora utilizzato
3) l'arco selezionato viene aggiunto al MST se la sua aggiunta non forma dei cicli nell'albero, altrimenti viene scartato
4) si ripete dal punto 2 fino a quando non gli archi non sono terminati

L'algoritmo di Kruskal è adatto alla generazione di labirinti poiché crea un MST che garantisce l'esistenza di una e una sola soluzione del labirinto e la connessione di tutti i nodi, cioè l'assenza di zone inaccessibili, una volta creato il MST allora viene scelto un nodo del grafo come casella di partenza e un secondo come casella di arrivo.

Ovviamente, sarebbe stato possibile utilizzare altri algoritmi diversi da quello di Kruskal per creare labirinti, come ad esempio l'algoritmo di Prim o altri algoritmi ricorsivi. L'algoritmo di Kruskal tuttavia presenta alcuni vantaggi: rispetto ad altri algoritmi, come quello di Wilson, non trattato a lezione, risulta essere sufficientemente facile da implementare, garantisce una complessità finale ridotta ed inoltre, aspetto dal carattere soggettivo e qualitativo, ma degno di nota, dà al labirinto generato un aspetto ramificato

e con molte strade secondarie, che aumentano il grado di difficoltà nella soluzione del labirinto stesso. Altri algoritmi invece tendono a creare labirinti lineari e con poche strade secondarie, che specialmente per labirinti di piccole dimensioni portano a soluzioni immediate.

Algoritmo di Kruskal, implementazione

	MST-KRUSKAL(G,w) // G grafo e w array dei pesi
1	$A=\emptyset$
2	**for** ogni vertice $v \in G.V$
3	**MAKE-SET**(v)
4	ordina gli archi di $G.E$ in senso non decrescente rispetto al peso
5	**for** ogni arco $(u,v) \in G.E$, preso in ordine di peso non decrescente
6	**if FIND-SET**(u)$\neq$**FIND-SET**(v)
7	$A=A\cup\{(u,v)\}$
8	**UNION**(u,v)
9	**return** A

Pseudocodice dell'algoritmo di Kruskal

L'implementazione tramite strutture Union-find (necessaria per migliorare l'efficienza dell'algoritmo) è possibile tramite le realizzazione di tre funzioni fondamentali:

MAKE-SET(v): crea un insieme il cui unico elemento è v. Dove v ne diventa un rappresentante e un riferimento.

FIND-SET(v): restituisce un riferimento all'insieme di cui v ne è rappresentante.

UNION (v, u): fonde gli insiemi di appartenenza dei due nodi u e v in un nuovo insieme formato dall'unione dei due precedenti.

Nel nostro caso l'algoritmo di Kruskal è stato definito come metodo di istanza all'interno della classe Graph. La classe Graph ha come sottoclasse la classe subset, che rappresenta gli oggetti delle strutture Union-find, ed ha come variabili di istanza parent e rank, entrambi di tipo int. L'esecuzione dell'algoritmo di Kruskal procede quindi in questo modo:

1) Inizialmente gli archi del grafo vengono ordinati tramite la funzione Arrays.sort( ), metodo definito da java, all'interno di java.util.Arrays, il metodo procede in questo modo:

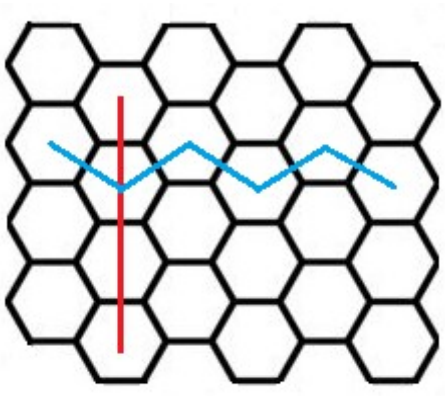
L'array da ordinare viene scomposto ricorsivamente secondo l'algoritmo Merge sort, questo procedimento va avanti finché l'array di partenza si ritrova a essere scomposto in molti sotto array di piccole dimensioni. Questi vengono quindi ordinati tramite l'algoritmo Insertion sort, per poi essere ricombinati in un unico array ordinato.
Gli oggetti di tipo Edge vengono ordinati in base alla variabile di istanza weight, è stato definito quindi un opportuno metodo CompareTo() all'interno della classe Edge, per poter stabilire quale è l'oggetto Edge di peso maggiore.

2) Per ogni nodo del grafo viene specificato l'insieme a cui fa riferimento, tramite la sottoclasse subset, questa permette di specificare, tramite la variabile parent, il riferimento dell'insieme di cui un nodo v ne è rappresentante.

3) Si procede con l'esaminare gli oggetti Edge ordinati, i metodi find() e union(), vengono utilizzati per poter determinare se un arco deve far parte del MST o meno, questi permettono a partire da un nodo, di determinare se fanno parte dello stesso insieme o meno, e di unire questi insiemi cambiando le variabili rank, e parent.

Sistema di coordinate per le celle del labirinto

L'algoritmo proposto prevede che si possano creare labirinti aventi celle a base esagonale o a base quadrata. In ciascuno dei due casi è necessario creare un valido sistema di riferimento per poter ricondurre ogni cella del labirinto a una specifica coordinata. Questo lavoro risulta particolarmente facile nel caso in cui le celle siano a base quadrata, qui la cella che si trova sulla riga i e la colonna j avrà coordinate (i,j). In verità anche per i labirinti a base esagonale vale un ragionamento molto simile: si definiscono in maniera molto semplice delle righe e delle colonne costituite da esagoni, nel modo sotto illustrato, anche qui l'esagono collocato sulla riga i e la colonna j avrà coordinata (i,j).



Nella figura sono evidenziate una riga e una colonna di una possibile griglia esagonale: in rosso la colonna 1,e in blu la riga 1.

Inizio esecuzione dell'algoritmo

L'algoritmo proposto chiede all'utente quattro informazioni:

1) il numero di righe del labirinto
2) il numero di colonne
3) un seme (in seguito spiegherò il motivo di questa richiesta)
4) il tipo di pianta del labirinto (1 se esagonale, 0 se quadrata)

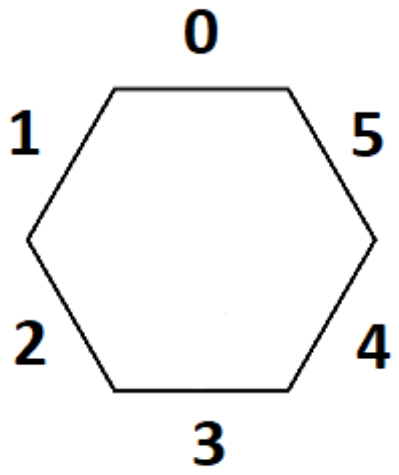
A questo punto viene creato un oggetto grafo con righe*colonne nodi e un numero di archi pari a due volte il numero dei nodi nei labirinti a base quadrata, o a tre volte il numero dei nodi, nei labirinti a base esagonale. Gli archi vengono poi posizionati in modo che tutte le celle adiacenti abbiano un arco che le collega, se il labirinto è a base quadrata, allora è facile capire che il quadrato di coordinate (i,j) confina con i quadrati di coordinate (i-1, j), (i+1,j), (i,j-1) e (i,j+1). Tuttavia la questione è leggermente più complessa se il labirinto risulta essere a base esagonale, qui si possono distinguere due casi:

1) se la colonna su cui si trova la cella è pari, l'esagono confina con quelli di coordinate: (i-1,j), (i-1,j-1), (i,j-1), (i+1,j), (i,j+1) e (i-1,j+1)
2) se la colonna su cui si trova la cella è dispari, l'esagono confina con quelli di coordinare: (i-1,j), (i,j-1), (i+1,j-1), (i+1,j), (i+1,j+1) e (i,j+1)

Una volta che tutte le caselle adiacenti sono state collegate, si da un peso casuale a ciascun arco, utilizzando il seme che è stato richiesto a inizio esecuzione dell'algoritmo. In questo modo, sul grafo ottenuto, è ora possibile applicare l'algoritmo di Kruskal.

Il problema della rappresentazione visiva

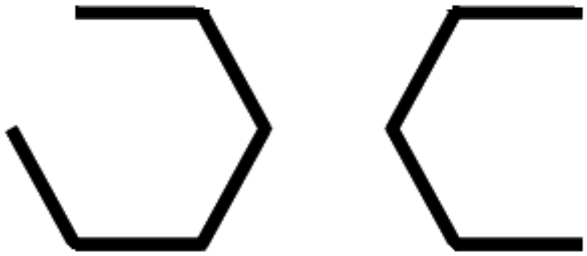
L'algoritmo di Kruskal, prende il grafo creato e calcola un possibile MST, restituendo un array di archi, il quale contiene tutti gli archi che sono stati scelti da Kruskal, dal grafo passatogli, per la creazione del MST. Ora il grafo ottenuto sarà poi in seguito visualizzato tramite un grafico SVG. Per arrivare a vedere, quindi, il labirinto che è stato appena creato, è necessario innanzitutto, nel caso di celle a base esagonale, numerare i lati di ogni esagono che compongono il MST da 0 a 5, nella maniera mostrata in figura.



Il lato dell'esagono numero 0 è quello “più alto”, per numerare i restanti cinque lati, si procede in senso antiorario.

Per ogni nodo x del Minimum Spanning Tree viene assegnato un numero compreso tra 0 e 63, che abbiamo chiamato peso del nodo x. Se un nodo ha come peso 0 vuol dire che esistono sei archi i quali permettono di collegare tale esagono con tutti gli altri esagoni adiacenti. Se un nodo ha invece come peso 63, al contrario, significa che in ogni lato dell'esagono è presente un muro. In generale il numero assegnato ad ogni nodo è così calcolato: inizialmente al nodo x viene assegnato il numero 63, poi, scorrendo la lista degli archi (cioè l'array di archi che ci ha restituito l'algoritmo di Kruskal), se un arco collega il nodo x con un secondo attraverso il lato i (ove i è ovviamente un numero compresa tra 0 e 5), allora si sottrae a 63 il numero 2^i; si continua quindi con questa logica, fino a che non finiscono tutti gli archi del MST, per poter eseguire questo compito sono state create degli appositi metodi all'interno della classe Labyrinth, ovvero findConnectionSquare e assignvaluesquare (per i labirinti a base quadrata). Al termine di tale processo ogni nodo avrà assegnato un numero da 0 a 63 che indica la configurazione di tale nodo. La matrice che contiene i numeri assegnati a ogni nodo del MST, servirà ora per la creazione del file SVG. Ovviamente queste considerazioni sono valide nel caso in cui le celle del labirinto siano esagonali, in caso contrario, cioè se

sono quadrate, si può fare comunque un ragionamento analogo: si numerano i lati da 0 a 3, e si calcola, per ogni nodo, un numero compreso tra 0 e 15 che indica la configurazione di tale nodo nel MST calcolato.

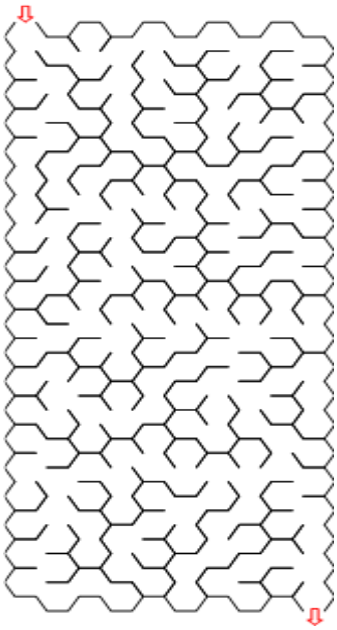


Due possibili esagoni del labirinto, il primo ha come peso 61, poiché sprovvisto solamente del lato 1 (63 – 2^1 = 61), il secondo ha come peso 15 poiché gli mancano i lati 5 e 4 (63 – 2^4 – 2^5 = 15).

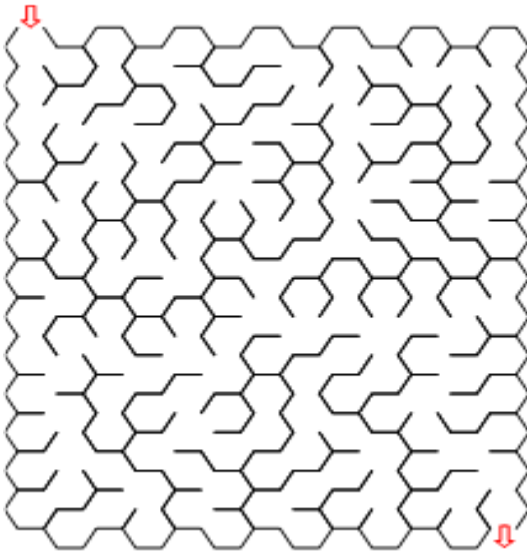
Grafico SVG del labirinto

Ora che ogni cella esagonale o quadrata del labirinto è stato numerata, è possibile risalire alla configurazione della cella stessa mediante il numero assegnatogli, il programma comunque, può comunicare all'utente sia gli archi del MST che i pesi dei nodi, (la possibilità di stampare il peso dei nodi e gli archi del MST è stata contrassegnata come commento all'interno del codice, in quanto comportava un grave decadimento delle prestazioni). A questo punto dell'esecuzione, quindi, il programma inizia a scrivere un file di testo che servirà per poter rappresentare il labirinto creato. Nel file di testo che verrà creato, sarà presenta una legenda che fa corrispondere ad ogni possibile peso di un nodo, una figura geometrica precisa. In questo modo per poter scrivere il file è necessario solamente esaminare le celle del labirinto prese una per volta, tramite un doppio ciclo for, leggere il peso di tale cella e usarlo per fargli corrispondere una precisa figura geometrica inoltre a ogni specifica figura geometrica, viene riservato uno specifico spazio, in modo tale che più figure non si sovrappongano nel risultato finale

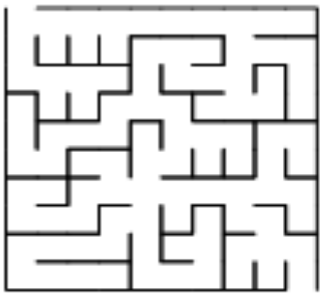
Possibili Labirinti



(rows = 13, cols = 13, seed = 1, type = 1)



(rows = 10, cols = 20, seed = 1, type =1)



rows = 10, cols = 10, seed = 200, type = 0

Complessità dell'algoritmo

L'analisi della complessità del programma può essere suddivisa in tre parti, indipendentemente dal tipo di cella utilizzata:

1) Creazione del grafo associato al labirinto
2) Applicazione dell'algoritmo di Kruskal al grafo
3) Disegno dell'algoritmo su file

Definisco: r=rows, c=cols, n=|V|=r*c, |E|=Θ(n)
|E|=Θ(n) in quanto ogni cella del labirinto è collegata solo alle celle adiacenti ad essa (4 per le celle quadrate e 6 per le celle esagonali).

La creazione del grafo avviene tramite due cicli for annidati, che si ripetono rispettivamente r e c volte. Tutte le istruzioni all'interno dei cicli hanno complessità Θ(1).
Si avrà quindi una complessità pari a Θ(r*c)=Θ(n).

L'algoritmo di Kruskal a sua volta può essere diviso in varie sezioni.
All'inizio si crea un array di Edge di lunghezza n. La sua inizializzazione richiede un tempo Θ(n).
Viene poi ordinato l'array edge di lunghezza |E| tramite la funzione Arrays.sort(). Questa funzione, come specificato nella documentazione ufficiale di Java, è una particolare implementazione dell'algoritmo TimSort, che a sua volta è una combinazione di MergeSort e InsertionSort.
Possiede quindi nel caso peggiore le caratteristiche di complessità di un algoritmo MergeSort, ovvero O(|E|*log|E|)=O(n*logn).
In seguito viene creato un array di subset di lunghezza n ed inizializzato. L'operazione ha complessità Θ(n) in quanto consiste in un solo ciclo for ripetuto n volte.
Infine si entra in un ciclo while ripetuto n-1 volte che contiene al suo interno chiamate ad altri due metodi: find e Union.

- La funzione find è una funzione ricorsiva che risale l'albero del subset specificato, restituendo la radice. Nel caso peggiore si dovrà quindi risalire di un numero di livelli pari all'altezza dell'albero, che nel caso peggiore sarà O(logn) in quanto la funzione Union è realizzata tramite union by rank, che impedisce all'altezza dell'albero di crescere in maniera lineare. La complessità del metodo find sarà quindi O(logn).
- Il metodo Union è composto da sole operazioni di complessità costante, fatta eccezione le chiamate alla funzione find. Avrà quindi complessità O(logn).

Il ciclo while avrà allora una complessità pari a (n-1)*O(logn)=O(n*logn).
In conclusione questa particolare implementazione dell'algoritmo di Kruskal avrà complessità pari a O(n*logn), che rispetta il limite teorico dell'algoritmo, pari a O(|E|*log|V|), raggiungibile infatti utilizzando strutte Union-find.

L'ultima parte del codice si occupa di disegnare il labirinto in un file di output, chiamando le funzioni assignValueSquare(Hex) e printSvgSquare(Hex).

- assignValueSquare(Hex)
  Genera un array bidimensionale di int di dimensioni r*c e lo inizializza: complessità Θ(n).
  Contiene poi un ciclo for ripetuto Θ(n) volte nel quale viene chiamata la funzione findConnectionSquare(Hex) che ha complessità Θ(1): complessità Θ(n).
- printSvgSquare(Hex)
  Questa funzione contiene solo istruzioni di complessità costante ad eccezione di un doppio ciclo for annidato ripetuto r*c=n volte.
  Avrà quindi complessità Θ(n).

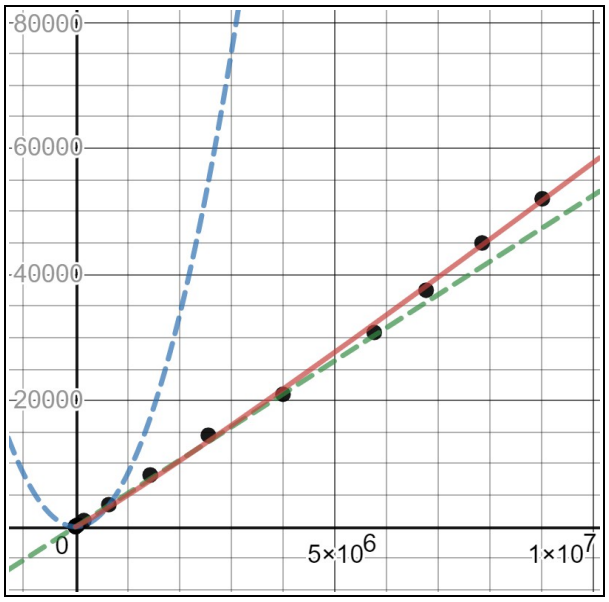
La scrittura del file di output avrà quindi complessità totale Θ(n).

L'intero programma avrà quindi una complessità pari a O(n*logn), determinata dal tempo necessario all'esecuzione dell'algoritmo di Kruskal. Le altre sezioni hanno infatti una complessità inferiore pari a Θ(n).

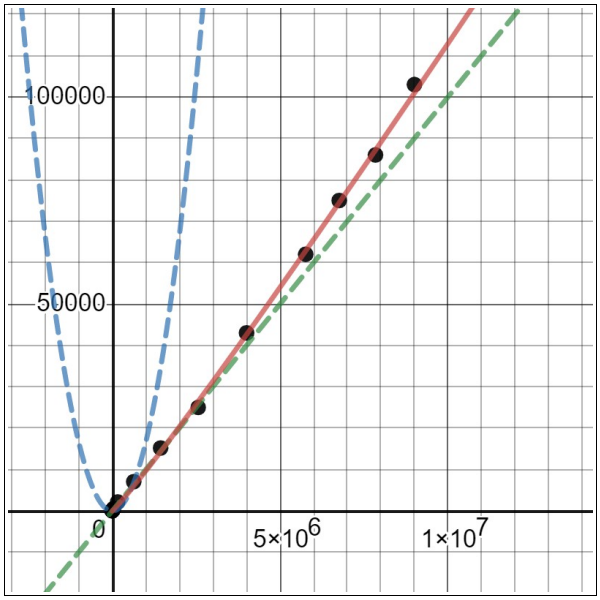
Queste osservazioni sono supportate anche dai dati sperimentali raccolti dall'esecuzione del programma stesso, che anche se non forniscono una prova decisiva per la correttezza della complessità calcolata, possono fornire una prova empirica che supporta le ipotesi fatte.
Sono stati quindi generati tramite il programma dei labirinti di varie dimensioni, sia quadrati che esagonali, e sono stati misurati i tempi di esecuzione tramite la funzione fornita da Java System.currentTimeMillis().

		Tempi (ms) - PC 1		Tempi (ms) - PC 2	
r x c	n	Quadrato	Esagono	Quadrato	Esagono
10x10	100	20	20	35	40
20x20	400	30	30	50	55
30x30	900	37	37	70	70
50x50	2500	55	55	100	105
60x60	3600	65	65	110	120
100x100	10000	110	118	220	230
200x200	40000	260	300	520	570
400x400	160000	980	1100	1930	2150
800x800	640000	3500	4000	6350	7070
1200x1200	1440000	8200	9000	13100	15200
1600x1600	2560000	14500	16000	23400	25000
2000x2000	4000000	21000	26000	35000	43000
2400x2400	5760000	30800	39200	51700	62000
2600x2600	6760000	37500	46500	61200	75000
2800x2800	7840000	45000	55000	73500	86000
3000x3000	9000000	52000	65500	85500	103000

PC 1 – Quadrato



PC 2 - Esagono



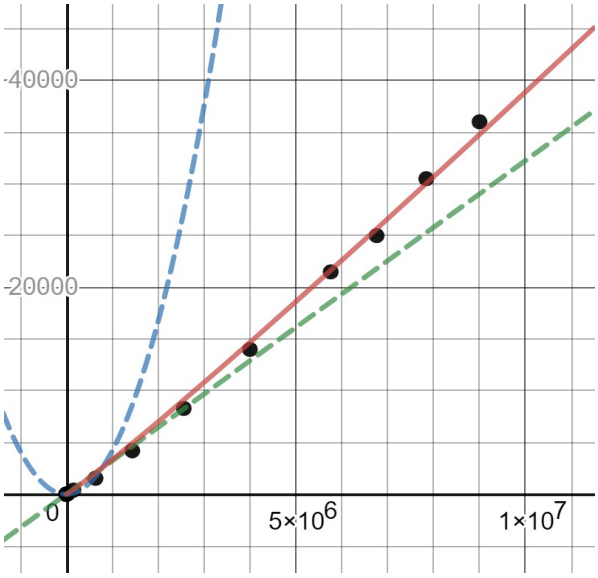
Nei due grafici il valore di n è stato posto sull'asse x e il tempo corrispondente sull'asse y, ed i punti della tabella sono contrassegnati in nero.
La curva azzurra è una curva del tipo $y=\Theta(x^2)$.
La curva verde è una retta del tipo $y=\Theta(x)$.
La curva rossa è una curva del tipo $y=\Theta(x*\log x)$.

Si può osservare come in entrambi i casi la curva azzurra e la curva verde approssimano in modo accettabile i punti con n piccola. Con l'aumentare del valore di n però la curva azzurra cresce molto più velocemente dei punti registrati e la curva verde cresce invece più lentamente.
La curva rossa, quella che si ipotizza approssimi asintoticamente in modo migliore i valori ottenuti dal programma, sembra effettivamente approssimare in modo sufficientemente accurato la crescita del tempo di esecuzione.

Si può inoltre notare come la curva rossa cresca molto lentamente, di poco più veloce della curva lineare verde, e quasi identicamente ad essa fino a valori di n di circa 10^6. Per valori piccoli di n si può quindi approssimare l'efficienza del programma ad una complessità quasi lineare.

Lo stesso procedimento di raccolta dei tempi di esecuzione è stato effettuato sull'algoritmo di Kruskal indipendentemente dal resto del programma.

r x c	n	Tempo (ms)
10x10	100	1
20x20	400	2
30x30	900	3
50x50	2500	7
60x60	3600	9
100x100	10000	17
200x200	40000	70
400x400	160000	400
800x800	640000	1550
1200x1200	1440000	4200
1600x1600	2560000	8300
2000x2000	4000000	14000
2400x2400	5760000	21500
2600x2600	6760000	25000
2800x2800	7840000	30500
3000x3000	9000000	36000



Anche in questo caso si possono fare le stesse considerazioni fatte con i grafici precedenti, il che supporta l'idea che l'algoritmo di Kruskal implementato abbia complessità O(n*logn).
Confrontando i dati della tabella precedente con quelli di quest'ultima, si può anche osservare come il rapporto tra il tempo di esecuzione dell'algoritmo di Kruskal e il tempo totale di esecuzione del programma cresca con l'aumentare di n. Questo fatto supporta ulteriormente l'ipotesi che la complessità totale del programma sia dovuta all'algoritmo di Kruskal e non alle altre sezioni. Infatti con l'aumentare di n il tempo utilizzato da Kruskal diventa sempre più vicino al tempo totale dell'esecuzione, ovvero cresca più velocemente rispetto alle altre parti del programma, che secondo le ipotesi fatte crescono in modo lineare.