Progettazione di Algoritmi a.a. 2013-2014

Federico Scozzafava Francescomaria Faticanti
Vincenzo de Pinto

2 luglio 2014

L'informatica non riguarda i computer più di quanto l'astronomia riguardi i telescopi

Edsger Wybe Dijkstra

Indice

1 Introduzione

L'obiettivo di questo progetto è quello di ideare ed implementare un *algoritmo di tracciamento* che calcoli le migliori linee di collegamento tra gli **elementi**[1] di un circuito.
L'algoritmo richiesto riceve in input la definizione del circuito in questione come una griglia di larghezza W e altezza H, l'insieme degli elementi e una lista di coppie di **punti**[2] da collegare.
Per ogni coppia di nodi appartenente alla lista di punti ricevuta in input, viene calcolato il costo del **cammino minimo**. Tale costo non è dato semplicemente dalla lunghezza del cammino, ma, per come è definito il problema, rappresenta una linea che collega i nodi dati minimizzando nell'ordine il numero di incroci, il numero di svolte ed infine la lunghezza.

Nel percorso di sviluppo delle varie *idee algoritmiche* che hanno portato alla soluzione finale si è rivelato decisivo tenere conto dell'imposizione facoltativa del ***vincolo di svolta***.
Il progetto ha richiesto inoltre un'accurata analisi dei requisiti e della complessità, in particolare al fine di scegliere strutture dati appropriate.
In fase di implementazione è emersa l'importanza della progettazione e della sperimentazione pratica degli algoritmi fino a quel momento affrontati prevalentemente dal punto di vista teorico.

Nel proseguo, dopo un breve *excursus* sulle prime elaborazioni dell'algoritmo e sulle relative criticità, viene illustrato l'algoritmo risolutivo (Capitolo 2), la sua implementazione finale come risultato di raffinamenti successivi (Capitolo 3) e l'analisi della complessità (Capitolo 4).
Infine sono riportati i dati relativi ai test condotti su diverse macchine (Capitolo 5).

[1] Un elemento, definito con una sequenza chiusa di punti del circuito, rappresenta un oggetto invalicabile sul circuito; ai bordi dell'elemento possono essere situati i punti di inizio e fine di ogni linea.

[2] Un punto del circuito è identificato da una coppia di coordinate (x, y) nella griglia, e quindi da un nodo nel grafo associato.

2 Idea Algoritmica

2.1 Prime idee

Inizialmente si è optato per la modifica di un algoritmo noto, che fosse in grado di operare su un grafo idealmente molto vicino alla rappresentazione della griglia del file di input.

A tale scopo sono state perseguite due idee basate rispettivamente su una visita BFS del grafo e su una ricerca dei cammini minimi mediante l'algoritmo di Dijkstra.

Visita BFS È stato impiegato l'algoritmo di *visita in ampiezza* basandosi sull'idea che, vedendo il grafo rappresentato come un reticolo, tutti gli archi avessero costo unitario.
Mettendo successivamente insieme idee provenienti *dall'algoritmo di Lee* ed *A**, si è giunti all'implementazione di una coda di priorità nella visita che estraesse i nodi di minor costo in funzione del numero di incroci, svolte e lunghezza del cammino.

Dijkstra È stato impiegato l'algoritmo di *Dijkstra* su un grafo inizialmente non pesato impostando dinamicamente a tempo di visita:

- il peso degli archi;
- il "nonno" del nodo corrente.

In questo modo si è tentato di esprimere il concetto di svolta.

Gli algoritmi proposti si sono dimostrati sin da subito non corretti: a fronte di numerosi casi particolari e controesempi non facili da valutare a priori, nessuna imposizione di ulteriori condizioni e controlli ha permesso di giungere ad una versione stabile.

Il calcolo di una svolta in un percorso comporta il tener traccia, per ogni nodo, del padre e del "nonno" di quest'ultimo nel grafo. Questa procedura di impostazione dei padri risulta complessa operando su una struttura a matrice: l'impostazione del padre del nodo, e quindi della svolta, è influenzata dall'ordine di visita del nodo. Questo fa sì che si giunga ad un risultato non sempre ottimale.

Dalla Figura 1 si può notare che, impostando il nodo **S** come nodo sorgente nell'algoritmo, gli adiacenti di questo possono essere raggiunti con costo unitario. A questo punto, per raggiugere il nodo **B** esistono due possibili cammini di egual costo: uno passante per il nodo **A** (Figura 2 (a)) e l'altro passante per il nodo **D** (Figura 2 (b)).

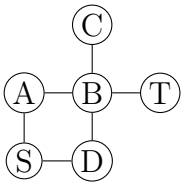

Figura 1: controesempio

Da ciò emerge come la posizione del nodo target rispetto ai nodi impostati fino a quel momento sia determinante nella scelte effettuate ad ogni passo dall'algoritmo: infatti, se il nodo target fosse il nodo **C**, il cammino migliore sarebbe quello passante per **D**, mentre se il nodo target fosse il nodo **T**, il cammino migliore sarebbe quello passante per **A**.

Poiché esistono due vie per raggiungere di egual costo il nodo **B**, il cammino verso il nodo destinazione potrebbe passare indifferentemente per **A** o per **D**, determinando per alcune istanze cammini non minimi. Ciò dimostra la non correttezza dell'algoritmo.

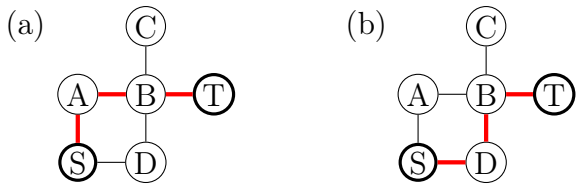

Figura 2: cammino minimo (a), cammino errato (b)

La ricerca di successive soluzioni si è spostata verso l'adozione di una *struttura dati* in grado di agevolare l'operazione di impostazione dei padri e la relativa gestione delle svolte.

Si è giunti, quindi, alla creazione di un grafo associato alla griglia data in input in grado di catturare le informazioni necessarie alla ricerca del cammino ottimale.
Spostare parte della complessità dall'algoritmo alla struttura e adattare quest'ultima all'algoritmo ha permesso di *ridurre notevolmente la complessità* del problema.

2.2 Idea finale

2.2.1 Costruzione del grafo

Considerando la rappresentazione della griglia di input come un grafo G non diretto, in cui i vertici sono i punti della griglia caratterizzati dalle loro coordinate, e gli archi sono determinati dalle adiacenze dei punti sulla griglia, si costruisce un grafo G' non diretto e pesato in cui ogni nodo in G è rappresentato da un gruppo di al più quattro nodi in G', secondo il ragionamento descritto di seguito.

Ad ogni nodo generico v in G corrisponde un tipo particolare di nodo – che chiameremo **BigNode** – in G', formato da una "quadrupla" di nodi semplici – che chiameremo **node** – : *v'-nord*, *v'-sud*, *v'-ovest*, e *v'-est*.

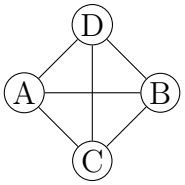

Figura 3: BigNode

Ogni *node* rappresenta un versante (nord, sud, est, ovest) di un $BigNode$, il punto di "passaggio" verso i suoi adiacenti nella direzione indicata. Questa rappresentazione, come verrà evidenziato meglio in seguito, appare efficace nell'espressione dei concetto di svolta ed incrocio.

I *node* componenti un $BigNode$ sono inoltre collegati da due archi "ortogonali"[3] di peso c ($\{N, S\}$, $\{O, E\}$) e quattro archi "diagonali"[4] di peso p ($\{N, O\}$, $\{N, E\}$, $\{S, O\}$, $\{S, E\}$.) tali che $p > c$ (Figura 3).

Se un nodo si trova ai bordi della griglia in G, in G' equivale ad una tripla di *node* sui lati, mentre corrisponde ad una coppia di *node* se si trova agli angoli (Figura 4).

Questa rappresentazione esprime il concetto di svolta nel grafo ed enfatizza il principio per cui svoltare applichi al cammino un costo maggiore rispetto a non cambiare direzione.

[3]Gli archi "ortogonali" rappresentano un'estenzione in lunghezza del cammino (o eventualmente un incrocio).

[4]Gli archi "diagonali" rappresentano una svolta nel cammino.

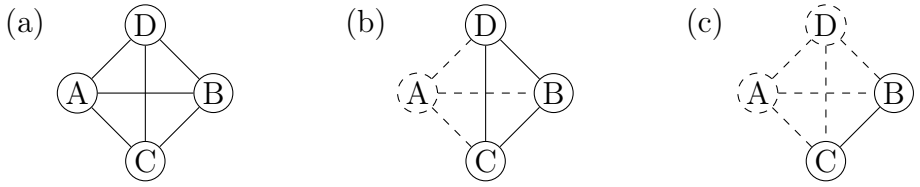

Figura 4: Tipi di *BigNode*: generico (a), di bordo (b), di angolo (c)

In Figura 5 è mostrato un esempio di trasformazione del grafo.

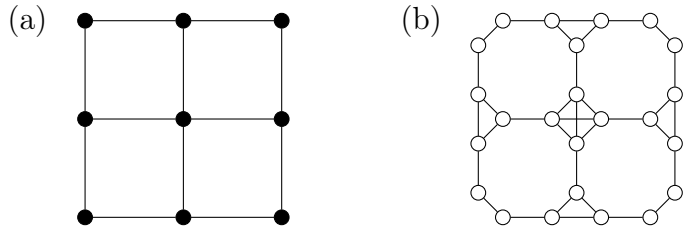

Figura 5: Griglia di input (a) e grafo associato (b)

2.2.2 Algoritmo di ricerca

L'algoritmo di Dijkstra viene eseguito sul grafo G' in modo da calcolare i cammini minimi dal nodo sorgente a tutti gli altri nodi del grafo.
Poiché ad un nodo v in G corrispondono quattro *node* in G', come punto di partenza è introdotto un nodo fittizio collegato, con archi di peso 0, ad ogni *node* del *BigNode* di partenza. Analogamente si procede per il nodo destinazione.
Dopo aver calcolato il percorso di costo minimo tra una coppia di nodi, si procede a ritroso nel grafo dal nodo destinazione al nodo sorgente, risalendo di padre in padre e sfruttando le informazioni contenute nel *vettore dei padri* costruito durante l'esecuzione dell'algoritmo.

Vengono ora riportate le fasi che compongono la procedura di *backtrace* (Figura 6) per la ricostruzione del cammino. Per ogni nodo appartenente alla sequenza si agisce in base alla seguente casistica:

- se si tratta del *BigNode* sorgente o del *BigNode* destinazione, si eliminano gli archi "diagonali" per imporre il *vincolo di svolta*[5], al

[5]Il *vincolo di svolta* impone che una linea non possa passare, e quindi né iniziare né terminare, in un nodo di svolta di un'altra linea.

fine di impedire che linee successive svoltino in corrispondenza dei nodi di partenza e arrivo;

- se il *BigNode* rappresenta una svolta per il cammino – ossia il cammino ha interessato un arco "diagonale" interno allo stesso – lo si elimina: in questo modo viene soddisfatto il *vincolo di svolta* per tutti i nodi del cammino;
- se il *BigNode* non rappresenta un punto di svolta – si tratta cioè di un nodo di incrocio o un nodo che estende in lunghezza il cammino senza cambio di direzione – viene coinvolto un arco "ortogonale" interno al *BigNode* e si effettuano i seguenti passaggi:
  - si eliminano i *node* agli estremi dell'arco facente parte del cammino, e quindi tutti gli archi incidenti, così da impedire ad una linea successiva di condividere un arco in G con la linea appena tracciata;
  - si incrementa di una costante k il peso dell'arco "perpendicolare" a quello interessato così da penalizzare un eventuale attraversamento in quel punto da parte di un cammino successivo.

  Queste operazioni garantiscono che, al momento del calcolo di un cammino successivo, l'algoritmo, preferendo archi di costo minimo, eviti – quando possibile – archi di peso maggiore o uguale a k, minimizzando incroci con altre linee preesistenti.

Al termine del procedimento descritto si eliminano i nodi "fittizi" aggiunti nella fase preliminare dell'esecuzione dell'algoritmo di Dijkstra.

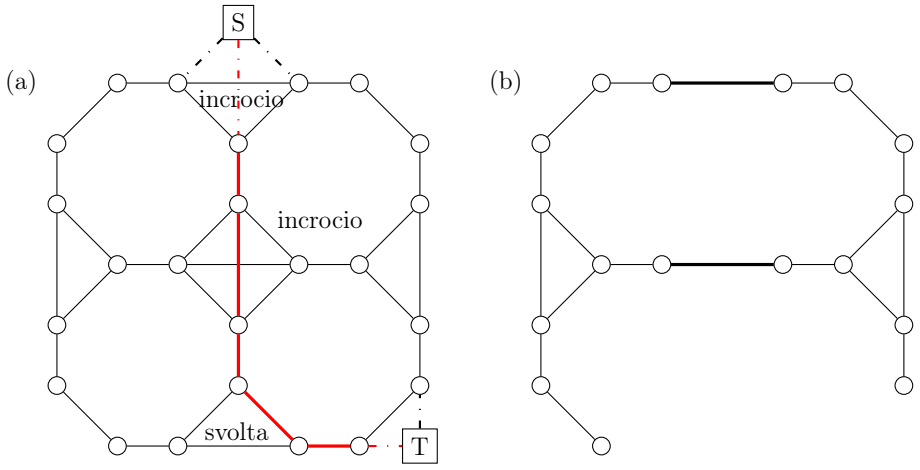

Figura 6: Tracciamento di una nuova linea (in rosso) prima (a) e dopo (b) l'applicazione della procedura di *backtrace*. In nero gli archi il cui costo è incrementato di un incrocio

3 Implementazione

Per la realizzazione del progetto è stato utilizzato il linguaggio Java.
La struttura dati prevalentemente utilizzata è stata l'array di interi. Questa struttura ha permesso una rappresentazione del grafo essenziale ed efficiente, garantendo un accesso in tempo costante agli elementi in esso contenuti. L'impiego dell'array nella rappresentazione del grafo sarà approfondito nella descrizione della versione 2.0 dell'algoritmo.

Si è scelto inoltre di implementare la coda dell'algoritmo di Dijkstra attraverso la *priority queue* di Java. Tale implementazione garantisce l'esecuzione in tempo di $O(log(n))$ delle operazioni di inserimento ed estrazione del minimo (offer, poll), in tempo lineare dei metodi *remove(Object)* e *contains(Object)* e in tempo costante dei metodi di recupero delle informazioni (peek e size).

3.1 Package e funzionalità

3.1.1 Main

Classe	Descrizione
MainClass	La classe responsabile della gestione di tutte le componenti del progetto

3.1.2 Core

Classe	Descrizione
Graph	La classe adibita alla costruzione di un grafo non diretto tramite liste di adiacenza
MyPriorityQueue	La classe rappresenta una coda di priorità basata sull'implementazione standard di *PriorityQueue* Java
Router	La classe adibita al calcolo del percorso di costo minimo tra una coppia di nodi usando un'implementazione dell'algoritmo di Dijkstra

3.1.3 Parser

Classe	Descrizione
InputParser	Classe responsabile della lettura e decodifica del file binario contenente i casi test
OutputParser	Classe responsabile della scrittura e codifica del file binario contenente i risultati dei casi test
Direction	Enum delle direzioni codificate nel file di input/output binario
Pair	Rappresenta un coppia di oggetti generici
Sequence	Rappresenta una sequenza iterabile di nodi del grafo come il contorno di una figura o una linea tracciata
ParsingException	Rappresenta un'eccezione nel parsing del file binario
Test	Rappresenta un caso test comprensivo di: altezza e larghezza della griglia, contorni delle figure, coppie di nodi da collegare

3.2 Versioni

v1.0 In questa versione iniziale il grafo è implementato come una matrice di oggetti di tipo **Node**. Ogni oggetto di tipo **Node** contiene come attributo una lista di oggetti di tipo **Edge**: la lista di adiacenza del nodo. Ogni elemento di tipo **Edge** rappresenta un arco ed ha come attributi due oggetti di tipo **Node**: gli estremi dell'arco. Questa implementazione risulta essere poco efficiente: basti pensare che per la costruzione di un grafo associato composto da 16 milioni di nodi il tempo impiegato è superiore ai cinque minuti.

v2.0 Nella versione 2.0 è stata introdotta una nuova rappresentazione del grafo. Poiché ogni $BigNode$ è costituito da 4 *node*, il grafo è rappresentato tramite un array di lunghezza $width * height * 4$. Per ricavare l'indice di ogni $BigNode$ in G' nell'array si utilizza la formula $(i * height + j) * 4$, dove i e j sono le coordinate del nodo in G.

Gli elementi dell'array sono la lista di adiacenza del *node*, rappresentata mediante un array di interi di lunghezza 5.

Il primo intero dell'array rappresenta il numero di adiacenti del *node*, in modo da ottimizzare la ricerca degli adiacenti e mantenere compatta la rappresentazione della lista.
Le successive 4 posizioni rappresentano gli eventuali adiacenti del nodo (infatti per ogni nodo si hanno al più quattro adiacenti).
Un adiacente è rappresentato da un intero (32 bit) codificato nel seguente modo:

- Il primo bit più significativo è vuoto per evitare cambi di segno.
- I successivi 3 bit codificano il peso dell'arco.

Rappresentazione	Descrizione
001	lunghezza unitaria del cammino
010	peso di un arco di svolta
100	peso di un arco di incrocio

- I 4 bit dal ventiquattro al ventisette non vengono impiegati.
- I 24 bit meno significativi codificano l'indice, nell'array principale, dell'adiacente del nodo. Considerando che nel caso peggiore si potrebbe avere una griglia di dimensioni 2000*2000, questo corrisponderebbe a dover indicizzare un array di 16 milioni di posizioni. Poiché con 24 bit si possono rappresentare 2^{24} (=16 777 216) numeri in base 10, il numero di bit scelto è sufficiente per codificare tutti i nodi anche nel caso peggiore.

$$31 \ \underbrace{30 \ 29 \ 28}_{peso} \ 27 \ 26 \ 25 \ 24 \ \underbrace{23 \ 22 \ \ldots \ 0}_{adiacente}$$

Nell'implementazione dell'algoritmo, il valore di priorità di ogni nodo (vettore delle distanze) è rappresentato da un valore di tipo *long* (64 bit). Un long è diviso in tre intervalli di 21 bit ciascuno (il bit più significativo viene ignorato per evitare cambi di segno): il primo intervallo rappresenta la lunghezza, il secondo intervallo rappresenta il peso delle svolte e il terzo il peso degli incroci.

$$63 \ \underbrace{62 \ \ldots \ 42}_{incroci} \ \underbrace{41 \ \ldots \ 21}_{svolte} \ \underbrace{20 \ \ldots \ 0}_{lunghezza}$$

v2.1 La rappresentazione del grafo rimane invariata rispetto alla versione 2.0. Grazie ad una modifica nell'implementazione dell'algoritmo di Dijkstra si giunge ad un notevole miglioramento nei tempi di elaborazione. La modifica riguarda l'inizializzazione del *min-heap*: invece di inserire nell'heap tutti i nodi del grafo ed inizializzarne i costi

fin da subito, si è optato per l'inserimento dei nodi dinamicamente via via che vengono "scoperti" tramie un loro adiacente. In altre parole, quando un nodo v viene estratto dall'heap ogni suo adiacente vi viene inserito con costo iniziale uguale alla somma del costo di v più il peso dell'arco che va da v al suo adiacente.

v2.2 Rispetto alla versione 2.1 l'utilizzo della memoria è ottimizzato evitando di riallocare per ogni coppia di nodi (per cui nella precedente versione era necessario calcolare un percorso di minimo costo) il vettore dei padri del grafo.

v2.3 Nella versione 2.3 viene impiegato anche un vettore dei nodi visitati che consente di distinguere i nodi il cui costo sia stato già stabilito. In questo modo se per un nodo v in G si ha che $visited[v] = true$, allora il nodo v non sarà preso in considerazione nelle successive iterazioni.

Definitiva Nella versione 2.3 la coda di priorità genera comportamenti non controllati dovuti alla possibilità che più copie di uno stesso nodo siano presenti nell'*heap* allo stesso tempo.
Quando un nodo viene rivisitato e il valore di priorità risulta essere minore di quello già impostato, il *Comparator* responsabile di effettuare i confronti tra gli elementi nella coda, poiché riferisce alle chiavi nel vettore delle distanze, non è in grado di tenere traccia del valore di priorità per le copie di uno stesso elemento, impostando per tutte il nuovo valore minimo.

Nella coda accade, quindi, che al momento dell'inseirmento di un nodo in seguito alla scoperta di un cammino di costo inferiore, la copia precedentemente inserita in posizione k determinata da una priorità iniziale p, si trovi ora in posizione k con un valore di priorità $s < p$.
La variazione di priorità di un nodo mantenendone invariata la posizione nel *min-heap* potrebbe alterare l'ordinamento degli elementi e quindi il risultato delle operazioni di inserimento ed estrazione del minimo.

Si è così giunti alla creazione di una **MyPriorityQueue** generica, basata sulla *PriorityQueue* di Java, che espone il metodo *offer(T e, long p)*, il quale permette di aggiungere un elemento e con priorità p. Incapsulare il valore di priortà nell'elemento – tramite l'uso di un oggetto wrapper interno alla classe – consente l'inserimento di molteplici copie di uno stesso oggetto senza modificare il valore di priorità per le copie preesistenti, mantenendo solida la rappresentazione dell'*heap*.

Questa modifica ha comportato un'ulteriore riduzione della complessità dovuta alla diminuzione degli accessi al vettore delle distanze, ora effettuati direttamente verso il campo interno all'oggetto esaminato.

4 Complessità

Sia n il numero di nodi del grafo G (dedotto dalla griglia presa in input) e sia m il numero di archi, poiché ogni nodo in G ha al più quattro adiacenti, possiamo affermare che, nel caso peggiore

$$m = \frac{\sum_{x \in V(G)} Dg(x)}{2} = 4n/2 = 2n. \tag{1}$$

Poiché nella rappresentazione del grafo G' ogni nodo in G è (nel caso generico) espresso con 4 nodi, una stima ragionevole del numero di nodi del grafo G è:

$$|V(G')| = 4|V(G)| = 4n \tag{2}$$

Una stima del numero di archi in G' (poiché ogni $BigNode$ in G' comprende al più sei nuovi archi) è invece:

$$|E(G')| = |E(G)| + 6n = 2n + 6n = 8n \tag{3}$$

La complessità della costruzione del grafo G' pertanto risulta essere dell'ordine di

$$O(|V(G')| + |E(G')|) = O(4n + 8n) = O(12n) = O(n) \tag{4}$$

cioè lineare nel numero di nodi.

Dalla complessità originale dell'algoritmo di Dijkstra ($O((n+m)log(n))$) si ottiene una complessità di:

$$O((|V(G')| + |E(G')|)log(|V(G')|)) = \\ O(8n * log(4n)) = O(8n(2 + log(n))) \sim O(nlog(n)) \tag{5}$$

Si osservi che la complessità dell'algoritmo di Dijkstra rappresenta un limite superiore abbastanza lasco per la complessità dell'implementazione poiché l'algoritmo termina non appena il nodo target viene estratto dalla coda (infatti quando il nodo viene estratto significa che il suo costo minimo è stato stabilito definitivamente).

Stando all'analisi sopra descritta, se per un dato caso di test si hanno p coppie di nodi di cui stabilire il cammino minimo, una stima per la complessità totale è: $p\ O(nlog(n))$.

5 Test

Le diverse versioni dell'algoritmo sono state testate utilizzando i file di input forniti in esempio.Tutti i test sono stati eseguiti con **Java 7** e **1gb di heap** massimo.

I test hanno permesso di valutare non solo la correttezza dell'algoritmo, ma anche l'efficienza in termini di memoria usata e tempi di esecuzione.

Le versioni testate sono state:

- **v2.0**
- **v2.1 (heapMod)** modifica implementazione heap
- **v2.2 (pMod)** ottimizzazione allocazione vettore dei padri
- **v2.3 (vMod)** ottimizzazione con vettore dei visitati
- **Definitiva** correzione problema heap

La versione 2.0 dell'algoritmo è molto meno efficiente delle successive, pertanto i tempi di esecuzione differiscono in modo consistente e sono difficilmente rappresentabili in un unico grafico.

A titolo di esempio il Grafico 1 riporta anche la colonna (blu) relativa al tempo di esecuzione dell'algoritmo nella versione 2.0 testato sulla macchina che si è rivelata più prestante. Come si può notare, già nell'Esempio 1, la cui esecuzione richiede pochi secondi, la differenza è sostanziale.

Negli altri grafici non vengono riportati i valori relativi all'esecuzione di questa versione che possono però essere letti nelle Tabella 1 dei dati (dove disponibili).

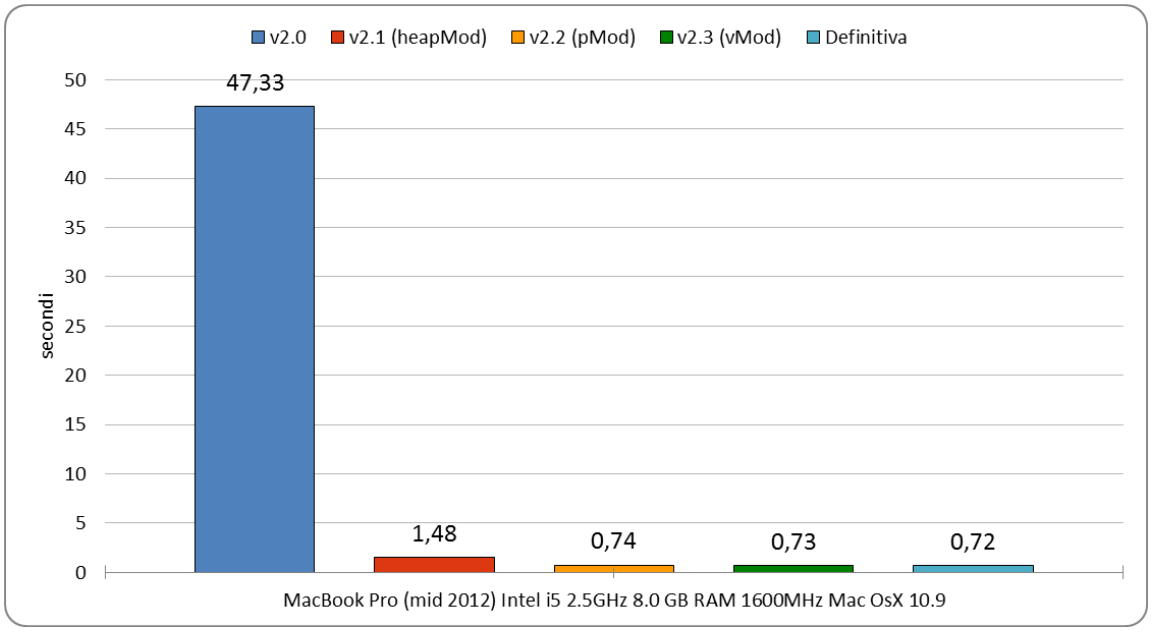

Grafico 1: Tempi di esecuzione dell'algoritmo (in secondi) nelle diverse versioni sull'Esempio 1, MacBook Pro.

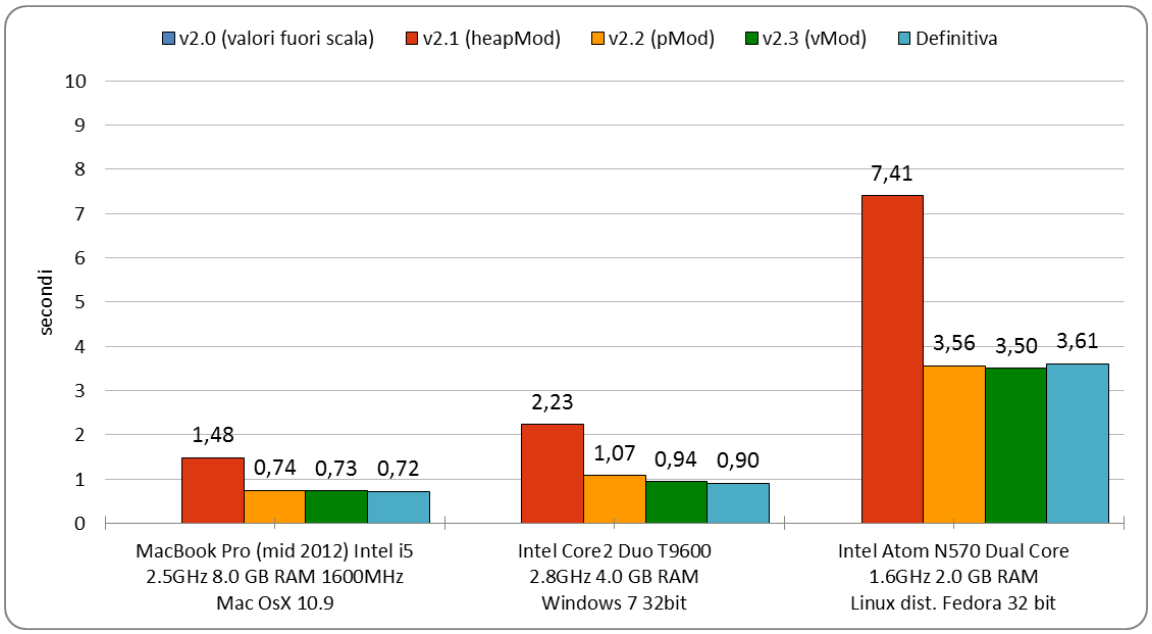

Grafico 2: Tempi di esecuzione dell'algoritmo (in secondi) nelle diverse versioni sull'Esempio 1.

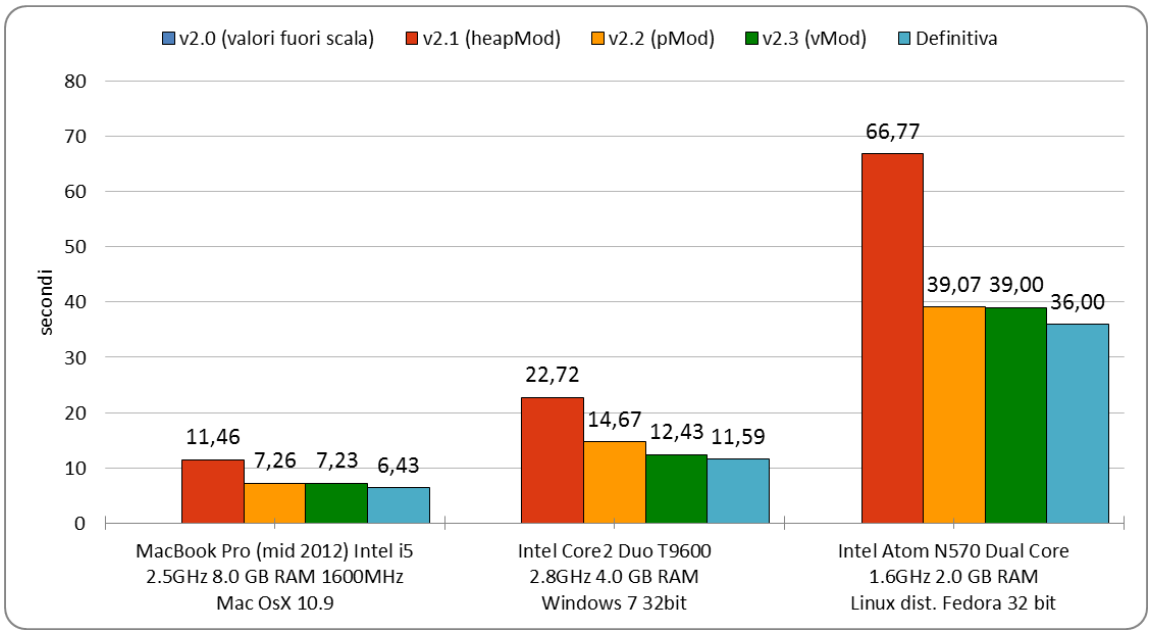

Grafico 3: Tempi di esecuzione dell'algoritmo (in secondi) nelle diverse versioni sull'Esempio 2.

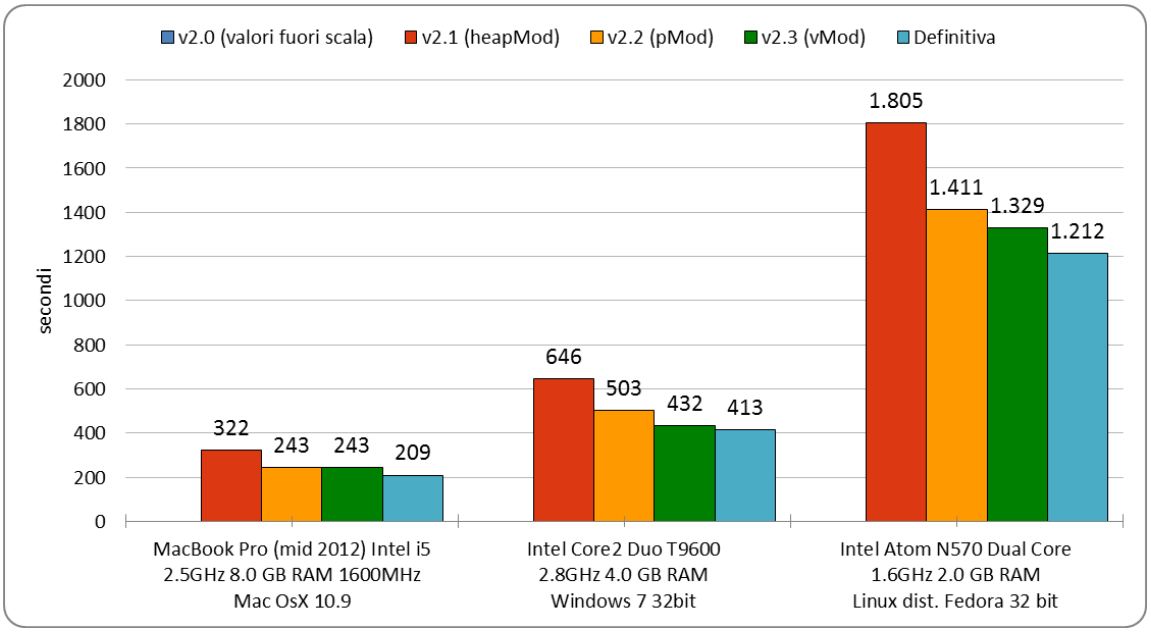

Grafico 4: Tempi di esecuzione dell'algoritmo (in secondi) nelle diverse versioni sull'Esempio 3.

Tutti i grafici evidenziano una notevole riduzione del tempo di esecuzione con l'ottimizzazione del vettore dei padri ottenuta nella versione 2.2 ed un ulteriore miglioramento, anche se di minore impatto, con le modifiche incluse nella versione definitiva.

Queste considerazioni valgono sia per i tre diversi Esempi che per ciascuna delle macchine sui si è scelto di condurre i test.

Esempio 1 (10 casi test)	v2.0	v2.1 (heapMod)	v2.2 (pMod)	v2.3 (vMod)	Definitiva
MacBook Pro (mid 2012) Intel i5 2.5GHz 8.0 GB RAM 1600MHz Mac OsX 10.9	47,33	1,48	0,74	0,73	0,72
Intel Core2 Duo T9600 2.8GHz 4.0 GB RAM Windows 7 32bit	245,82	2,23	1,07	0,94	0,90
Intel Atom N570 Dual Core 1.6GHz 2.0 GB RAM Linux dist. Fedora 32 bit	1.329,45	7,41	3,56	3,50	3,61

Esempio 2 (20 casi test)	v2.0	v2.1 (heapMod)	v2.2 (pMod)	v2.3 (vMod)	Definitiva
MacBook Pro (mid 2012) Intel i5 2.5GHz 8.0 GB RAM 1600MHz Mac OsX 10.9	2.822,19	11,46	7,26	7,23	6,43
Intel Core2 Duo T9600 2.8GHz 4.0 GB RAM Windows 7 32bit	ND	22,72	14,67	12,43	11,59
Intel Atom N570 Dual Core 1.6GHz 2.0 GB RAM Linux dist. Fedora 32 bit	ND	66,77	39,07	39,00	36,00

Esempio 3 (50 casi test)	v2.0	v2.1 (heapMod)	v2.2 (pMod)	v2.3 (vMod)	Definitiva
MacBook Pro (mid 2012) Intel i5 2.5GHz 8.0 GB RAM 1600MHz Mac OsX 10.9	ND	321,95	242,54	242,65	209,34
Intel Core2 Duo T9600 2.8GHz 4.0 GB RAM Windows 7 32bit	ND	646,30	503,05	432,42	413,12
Intel Atom N570 Dual Core 1.6GHz 2.0 GB RAM Linux dist. Fedora 32 bit	ND	1.805,08	1.410,58	1.329,45	1.212,32

Tabella 1: riepilogo dei tempi di esecuzione dell'algoritmo (in secondi) nelle diverse versioni. (ND: valore non disponibile)