Agenzia nazionale per le nuove tecnologie, l'energia e lo sviluppo economico sostenibile

MINISTERO DELLO SVILUPPO ECONOMICO

Ricerca di Sistema elettrico

Sviluppo di algoritmi per l'ottimizzazione del processo di elettrificazione di reti di trasporto pubblico urbano attraverso l'analisi delle possibili configurazioni di ricarica su sottoinsiemi della rete di trasporto

Giuseppe Francesco Italiano

Università di Roma Tor Vergata

Report RdS/PAR2016/229

SVILUPPO DI ALGORITMI PER L'OTTIMIZZAZIONE DEL PROCESSO DI ELETTRIFICAZIONE DI RETI DI TRASPORTO PUBBLICO URBANO ATTRAVERSO L'ANALISI DELLE POSSIBILI CONFIGURAZIONI DI RICARICA SU SOTTOINSIEMI DELLA RETE DI TRASPORTO

Giuseppe Francesco Italiano

Settembre 2017

Report Ricerca di Sistema Elettrico

Accordo di Programma Ministero dello Sviluppo Economico - ENEA
Piano Annuale di Realizzazione 2016
Area: "Efficienza energetica negli usi finali elettrici e risparmio energia negli usi finali elettrici ed interazione con altri vettori elettrici"
Progetto: "Mobilità elettrica sostenibile"
Obiettivo: "Scenari di mobilità elettrica", sub-obiettivo "a1:Strumenti di supporto TPL".
Responsabile del Progetto: ing. Antonino Genovese, ENEA

Il presente documento descrive le attività di ricerca svolte all'interno dell'Accordo di collaborazione "Sviluppo di algoritmi per l'ottimizzazione del processo di elettrificazione di reti di trasporto pubblico urbano attraverso l'analisi delle possibili configurazioni di ricarica su sottoinsiemi della rete di trasporto"
Responsabile scientifico ENEA: Massimo Celino
Responsabile scientifico Università degli studi di Roma Tor Vergata: Giuseppe Francesco Italiano

Indice

SOMMARIO 5
1 INTRODUZIONE 6
2 BACKGROUND TEORICO 8
2.1 PROGETTAZIONE DI PROGRAMMI PARALLELI 8
2.1.1 Capire il problema e il programma da realizzare 8
2.1.2 Partizionare il problema 8
2.1.3 Gestire le comunicazioni 8
2.1.4 Sincronizzare i task 8
2.1.5 Bilanciare il carico 8
2.2 BILANCIAMENTO DEL CARICO 9
2.2.1 Bilanciamento statico 9
2.2.2 Bilanciamento dinamico 9
3 FORMALIZZAZIONE DEL PROBLEMA DI OTTIMIZZAZIONE DEL TRASPORTO PUBBLICO 10
3.1 FUNZIONE OBIETTIVO E VINCOLI 10
3.1.1 Premessa 10
3.1.2 Modello matematico 10
3.2 LO SPAZIO DELLE SOLUZIONI 11
3.3 DESCRIZIONE DELLA RETE DI INPUT USATA COME DATASET DI INPUT 11
4 PROGETTAZIONE DI ALGORITMI PARALLELI PER L'ESPLORAZIONE ESAUSTIVA DELLO SPAZIO DI SOLUZIONI 13
4.1 ALGORITMO DI ESPLORAZIONE DELLA RETE 13
4.1.1 Metodo di Branch and bound 14
4.1.2 Comunicazione tra master e slaves 16
4.2 ALGORITMI DI BILANCIAMENTO 17
4.2.1 Bilanciamento di linee equo 17
4.2.2 Bilanciamento di linee first less, last more 19
4.2.3 Bilanciamento fisso in profondità 20
Implementazione dell'assegnamento di n-uple 22
4.2.4 Bilanciamento variabile in profondità 22
5 TECNOLOGIA SVILUPPATA 24
5.1 METODOLOGIE USATE 24
5.1.1 MPI e C++ 24
5.2 PROGRAMMA CON LOAD BALANCING STATICO 24
5.2.1 Struttura del programma 24
5.2.2 Manuale utente 26
6 TEST PRELIMINARI E VALUTAZIONI SPERIMENTALI 29
6.1 TEST AL VARIARE DEL TIPO DI BILANCIAMENTO 30
6.1.1 Utilizzo di 16 cores 30
6.1.2 Sottoreti assegnate e analizzate da ogni processore 30
6.1.3 Utilizzo di 32 cores 33
6.2 TEST AL VARIARE DEL NUMERO DI LINEE 35
6.2.1 Utilizzo di budget medio-grande e 32 cores 35
6.2.2 Utilizzo di budget piccolo e 256 cores 35
7 TEST PER IL CONFRONTO CON L'EURISTICA 37
7.1 TEST SU ISTANZE AD HOC 37
7.1.1 Istanza con 30 Linee e budget 30 37
7.1.2 Istanza con 30 Linee budget 60 38

7.1.3 Istanza con 30 Linee e budget 90 .... *38*
7.1.4 Analisi dei risultati di test fatti su istanze ad hoc *38*
7.2 TEST SUL CASO REALE DELLA RETE DI FIRENZE 39
8 CONCLUSIONI 43
8.1 SVILUPPI FUTURI 43
9 RIFERIMENTI BIBLIOGRAFICI 44
10 ABBREVIAZIONI ED ACRONIMI 45

Sommario

Questo lavoro di ricerca ha come obiettivo quello dell'ottimizzazione delle soluzioni per l'elettrificazione delle reti di trasporto pubblico. In questo ambito, l'attività dell'Università degli Studi di Tor Vergata si è concentrata sulla progettazione, analisi e ingegnerizzazione di algoritmi che siano in grado di trovare la soluzione ottima, esplorando lo spazio delle possibili soluzioni in modo parallelo e sfruttando l'infrastruttura di supercalcolo CRESCO per il calcolo scientifico ad alte prestazioni disponibili all'interno dell'ENEA.

Le sfide principali che si sono affrontate sono state la parallelizzazione del calcolo e il bilanciamento del carico di lavoro tra le molteplici risorse. In particolare, la creazione di un algoritmo di esplorazione dello spazio delle soluzioni e l'ideazione di algoritmi di bilanciamento sono state due delle attività principali su cui si è posto maggiormente l'attenzione.

Inoltre, grazie alla presenza del vincolo di budget, si è implementata una tecnica di branch and bound delle soluzioni, in grado di effettuare tagli di porzioni di rete che si è sicuri non conterranno soluzioni ottime. Questo ha permesso di poter analizzare reti di dimensioni significative.

Sono stati implementati quattro algoritmi di bilanciamento diversi. Ognuno di questi è una versione ottimizzata e modificata del precedente. Si tratta di algoritmi che cercano di predire il carico di lavoro per poterlo distribuire tra i vari processori attraverso un meccanismo di bilanciamento statico, la cui complessità risiede nel fatto che durante la fase di bilanciamento non si è a conoscenza delle porzioni di rete che ciascun processore esplorerà (in quanto le porzioni dello spazio di soluzioni assegnate possono essere soggette a dei tagli).

La differenza principale degli algoritmi implementati risiede proprio nella loro capacità di distribuire il carico. I primi due algoritmi non riescono a distribuire il workload in maniera equa e a sfruttare tutte le risorse di calcolo a disposizione. Invece, il risultato migliore lo si è raggiunto con gli ultimi due algoritmi, e in particolare con il bilanciamento variabile in profondità, in grado di partizionare il carico su di un numero di processori qualsiasi e ottenere una distribuzione del workload equa, a meno di fluttuazioni del carico impredicibili. Infatti, ogni processore, negli esperimenti che si sono eseguiti esplora un numero di sottoreti dello stesso ordine di grandezza di tutti gli altri processori.

Grazie a tali algoritmi di bilanciamento e alla tecnica del branch and bound implementata, si sono riuscite ad analizzare reti composte anche da 130 linee, e si sono ottenuti risultati interessanti anche sulla rete di Firenze costituita da 85 linee. Tali risultati sono sorprendenti se si pensa che lo spazio delle soluzioni di una rete con 100 linee è pari a $5.153 * 10^{47}$.

Infine, aver realizzato un algoritmo in grado di trovare sempre le migliori soluzioni ha reso possibile il confronto con l'algoritmo euristico implementato dall'Università degli studi Roma Tre. Inoltre, si è dimostrato che il passaggio dal sistema di alimentazione tradizionale a quello elettrico, risulta essere vantaggioso nel caso dell'istanza reale di Firenze.

1 Introduzione

L’accordo di collaborazione tra ENEA e il dipartimento di Ingegneria Civile ed Ingegneria Civile (DICII) dell’Università di Roma Tor Vergata ha come obiettivo l’ottimizzazione delle soluzioni per l’elettrificazione delle reti di trasporto pubblico. In particolare, l’Università contribuisce allo sviluppo di modelli di ottimizzazione attraverso l’analisi delle sottoreti di trasporto nelle differenti configurazioni di ricarica dei veicoli.

Con la realizzazione di questi modelli si vuole verificare la fattibilità tecnica ed economica dell’alimentazione elettrica autonoma, tramite opportune tecnologie di ricarica, per il trasporto pubblico su gomma di una città di dimensioni significative. Inoltre, tale modello consente di capire se la soluzione elettrica risulta economicamente vantaggiosa rispetto alle alternative convenzionali e di individuare le linee di trasporto maggiormente adatte alla trasformazione in alimentazione elettrica.

L’Università degli Studi di Tor Vergata si è concentrata sulla progettazione, analisi e ingegnerizzazione di algoritmi che siano in grado trovare la soluzione ottima, esplorando lo spazio delle possibili configurazioni di ricarica su sottoinsiemi opportuni delle linee della rete del trasporto pubblico considerata. Gli algoritmi sono stati implementati sull’infrastruttura di supercalcolo CRESCO (Computational RESearch centre on Complex system) per il calcolo scientifico ad alte prestazioni HPC (High Performance Computing) disponibili all’interno dell’ENEA.

Durante il periodo di progetto si sono sviluppate strategie di analisi di reti di trasporto pubblico con il fine di realizzare ricerche esaustive di reti e si è implementato un codice numerico parallelo utilizzato per la numerazione di sottoreti per il TPL (trasporto pubblico locale).

Risolvere un problema che va alla ricerca delle soluzioni ottime può non essere così semplice. Per trovare l’ottimo spesso si deve esplorare un insieme elevato di soluzioni, e ciò può richiedere un grande tempo di calcolo.

Per realizzare un software applicativo che calcoli la soluzione esatta, e che sia applicabile anche a istanze di dimensioni significative, si ha bisogno di poter realizzare e progettare applicazioni in grado di parallelizzare il calcolo e di bilanciare il carico di lavoro tra molteplici risorse di calcolo.

La principale sfida di questo lavoro è stata proprio assicurare un load-balancing efficiente, relativamente allo spazio di soluzioni da esplorare per trovare la soluzione esatta al problema di ottimizzazione del trasporto. Per risolvere questo problema è necessario capire, dato un budget di finanziamento destinato agli investimenti iniziali, quali linee di autobus conviene elettrificare per prime e con quali architetture di ricarica.

L'obiettivo di questa applicazione software è quello di riuscire a calcolare la soluzione esatta per questo problema di ottimo. La realizzazione di questo progetto consente di avere un nuovo strumento di confronto per le diverse euristiche che sono state implementate e che si implementeranno in futuro.

Per ottenere la soluzione migliore, si deve usare un approccio di tipo ricerca esaustiva, in grado di considerare tutte le possibili soluzioni ammissibili, al fine di trovare quella ottima. Tuttavia, lo spazio delle soluzioni del problema è tale che, volendo esaminare una rete con L linee elettrificabili con al più due tipi di architetture, si dovrebbero esaminare:

$$\sum_{k=1}^{L} \binom{L}{k} * 2^k \text{ soluzioni}$$

Per comprendere la portata di questo valore, basti pensare che per analizzare una rete con 100 linee si dovrebbero esplorare ben $5.153 * 10^{47}$ soluzioni! Questo numero ci fa comprendere che analizzare uno spazio di soluzioni così grande in tempi ragionevoli e con una semplice architettura di elaborazione (come ad esempio un PC) è impensabile.

Per tale motivo si è implementato un algoritmo in grado di:

- Esplorare lo spazio delle soluzioni in modo parallelo.
- Esplorare ogni possibile soluzione una e una sola volta, facendola esplorare ad uno e un solo processore.
- Sfruttare al meglio le risorse hardware di CRESCO.
- Effettuare tagli di porzioni di rete che non conterranno soluzioni ottime.

Quest'ultimo obiettivo è perseguibile con tecniche di branch and bound, che si sono potute implementare grazie alla presenza di vincoli di budget. Infatti, se una sottorete eccede il vincolo di budget, non ha senso andare ad esplorare le reti che contengono tale sottorete, perché non farebbero parte delle soluzioni ammissibili.

In questo lavoro si è implementato un algoritmo di esplorazione dello spazio delle soluzioni in modo parallelo, per poi focalizzarsi soprattutto sul load-balancing di tale applicazione.

Nei prossimi capitoli si analizzeranno tali algoritmi, specificando le metodologie usate, il funzionamento dell’applicazione software che si è realizzata e i risultati ottenuti dai test svolti.

2 Background teorico

2.1 *Progettazione di programmi paralleli*

La progettazione di programmi paralleli ha da sempre rappresentato una grande sfida. Quando si pensa a questo tipo di programmazione la responsabilità principale (se non unica) è del programmatore, che deve identificare e implementare il parallelismo [1]. Per far ciò si devono eseguire diverse attività, che per comodità verranno dettagliate nei seguenti sottoparagrafi.

2.1.1 Capire il problema e il programma da realizzare

Per prima cosa si deve analizzare il problema che si intende affrontare, per poter capire se si potrà sviluppare un programma parallelo. Per far ciò si deve:

- Identificare gli "hot spot" del programma, ossia le parti dove si spende la maggior parte del tempo.
- Identificare i colli di bottiglia nel programma.
- Identificare i fattori che inibiscono il parallelismo del codice, come la dipendenza dai dati.
- Valutare diversi tipi di algoritmi per poter capire quale sia il più adatto a sfruttare il parallelismo.

2.1.2 Partizionare il problema

È importante riuscire a decomporre il problema in porzioni di lavoro (task) discreti. Ci sono due modi principali per decomporre il carico di lavoro:

- *Domain decomposition*: Sono i dati associati al problema che vengono suddivisi tra i diversi task.
- *Functional decomposition*: È l'insieme di istruzioni del programma ad essere decomposto in vari task.

2.1.3 Gestire le comunicazioni

Il grado di comunicazione tra i task dipende dal tipo di problema. Ci sono problemi che vengono denominati imbarazzantemente paralleli, perché non necessitano di dover far comunicare i diversi task tra loro. Tuttavia, la maggior parte delle applicazioni non sono così semplici e richiedono che i task si scambino dati tra loro.
È bene sottolineare che la comunicazione:

- Implica un overhead di comunicazione.
- Ha bisogno di una sincronizzazione tra i task, i quali possono rimanere in attesa di una comunicazione, piuttosto che effettuare del lavoro utile.
- Può anche saturare la banda di rete disponibile fra i nodi.

2.1.4 Sincronizzare i task

Gestire l'esecuzione delle sequenze di lavoro e dei task è una parte critica della progettazione della maggior parte dei programmi paralleli. Infatti, può essere un fattore significativo che incide sulle prestazioni del programma. A tal proposito esistono diversi tipi di sincronizzazione come barrier, lock e semafori.

2.1.5 Bilanciare il carico

Il bilanciamento del carico si riferisce alla pratica di distribuire approssimativamente in maniera equa la quantità di lavoro tra i task, così da minimizzare il tempo in cui i task si trovano inattivi (in stato di idle). L'importanza del load balancing è dovuta allo stretto legame che c'è con le prestazioni del programma parallelo. Solitamente, migliore è il bilanciamento e migliori saranno le prestazioni. I modi di effettuare il load balancing sono principalmente due:

- Partizionare in modo statico ed equamente il carico di lavoro che ogni task riceve.
- Effettuare un assegnamento dinamico del lavoro. Infatti, quando il carico di lavoro che ogni task esegue non può essere previsto, è utile implementare uno *scheduler-task pool*. L'idea è che quando un task completa il proprio lavoro, riceve un nuovo compito da svolgere.

2.2 Bilanciamento del carico

Il bilanciamento del carico è la parte che più richiede l’intervento del programmatore. Per questo motivo si vuole approfondire la differenza tra bilanciamento statico e dinamico [3].

2.2.1 Bilanciamento statico

In questo approccio il bilanciamento del carico è ottenuto attraverso le informazioni preliminari che si hanno sul sistema. Ciò che un nodo dovrà calcolare viene determinato all'inizio dell'esecuzione, e una volta che viene assegnato il lavoro da eseguire ad un nodo non ci saranno comunicazioni finché il nodo non avrà completato il calcolo che gli è stato assegnato. Dunque, il workload viene assegnato a priori, senza poi tenere conto del carico corrente. Si tratta di metodi di bilanciamento non preemptive, cioè quando il carico viene assegnato ad un nodo non può essere trasferito ad un altro. Questo metodo riesce ad evitare l'overhead di comunicazione. D'altra parte, l'inconveniente principale di questo approccio è che mentre si effettuano le decisioni di assegnamento, non si tiene conto dello stato futuro del sistema. Questo può avere un grande impatto sulle prestazioni complessive del sistema a causa di fluttuazioni di carico nel sistema.

2.2.2 Bilanciamento dinamico

Il bilanciamento dinamico si basa su algoritmi in grado di monitorare i cambiamenti sul carico di lavoro del sistema e ridistribuire il lavoro di conseguenza. Solitamente gli algoritmi di bilanciamento del carico dinamico sono composti da tre policy:

- La policy di *trasferimento* stabilisce quali task sono idonei per essere trasferiti ad altri nodi.
- La policy di *locazione* stabilisce su quale nodo è meglio trasferire le operazioni previste dal task.
- La policy di *informazione* è il centro informativo per l'algoritmo di bilanciamento del carico.

Inoltre, gli algoritmi dinamici possono assumere tre diverse forme di controllo: centralizzato, distribuito o semi-distribuito.

Nella distribuzione centralizzata del carico, un nodo singolo (il nodo centrale) è nominato per essere responsabile di tutta la distribuzione del carico nella rete. Nella modalità distribuita la responsabilità è divisa fra tutti i nodi allo stesso modo.

Nella modalità semi-distribuita la rete dei nodi è suddivisa in cluster, dove ogni cluster è centralizzato. Il bilanciamento del carico dell'intero sistema viene raggiunto attraverso la cooperazione dei nodi centrali di tutti i cluster.

3 Formalizzazione del problema di ottimizzazione del trasporto pubblico

L'obiettivo di questo lavoro è trovare le migliori sottoreti (insieme di linee di bus) da elettrificare, sapendo di avere un budget limitato per far ciò. Il programma esplorerà tutte le linee di autobus attraverso una ricerca esaustiva in grado di considerare tutte le possibili combinazioni di sottoreti e tutti i possibili modi di elettrificare tali sottoreti.

3.1 Funzione obiettivo e vincoli

3.1.1 Premessa

Per poter comprendere la modellizzazione del problema è giusto fare una breve panoramica a riguardo dei trasporti pubblici su gomma. Una rete di TPL può essere vista come un insieme di linee, ed ognuna di esse avrà un determinato numero di autobus che è in grado di soddisfare l'intero servizio giornaliero.

Per il nostro obiettivo, una linea (*L*) è in realtà vista come un insieme di costi: i costi di investimento (*I*) e i costi operativi (*C*). In particolare:

- Nel caso del trasporto tradizionale (*T*) si hanno:
  - IL_l^T Costo di investimento di linea per il caso di alimentazione tradizionale (diesel).
  - CL_l^T Costo operativi di linea per il caso di alimentazione tradizionale (diesel).
- Nel caso del trasporto elettrico (*E*) si hanno:
  - IL_{la}^E Costo di investimento di linea.
  - CL_{la}^E Costo operativo di linea.
  - IN_{la}^{DE} Costo di investimento di nodo deposito.
  - CN_{la}^{DE} Costo operativo per il nodo deposito.
  - IN_{la}^{CE} Costo di investimento per i nodi capolinea.
  - CN_{la}^{CE} Costo operativo per i nodi capolinea.
  - IN_{la}^{FE} Costo di investimento per i nodi fermata.
  - CN_{la}^{FE} Costo operativo per i nodi fermata.

Inoltre, nel caso elettrico è da specificare che si può decidere di elettrificare una linea utilizzando tre tipi diversi di architetture (*a*) di autobus:

1. Tipologia *A*: autobus che necessitano di ricarica della batteria solo a deposito (*D*).
2. Tipologia *B*: autobus che necessitano di ricarica della batteria a deposito e capolinea (*C*).
3. Tipologia *C*: autobus che necessitano di ricarica della batteria a deposito, capolinea e alle fermate (*F*).

3.1.2 Modello matematico

Il programma che si è realizzato vuole risolvere un problema di ottimizzazione: dato un budget di finanziamento destinato agli investimenti iniziali (acquisto di veicoli e infrastrutture di ricarica), quale sottorete della rete complessiva di servizio conviene elettrificare e quale tipologia di ricarica scegliere per ogni linea?

La funzione obiettivo (si veda l'eq. 3.1) è quella di massimizzare la differenza (intesa come guadagno) tra i costi complessivi del trasporto tradizionale e i costi complessivi del trasporto elettrico.
L'unico vincolo a cui si è soggetti è il vincolo di budget disponibile per gli investimenti iniziali (si veda l'eq. 3.2). Infatti, si vuole che i costi di investimento per il trasporto elettrico siano compresi tra il budget massimo (I^M) e una percentuale di questo valore ($tol \cdot I^M$).

Ne segue che il modello matematico del problema può essere scritto nel seguente modo:

Funzione obiettivo (*eq. 3.1*):

$$\max \sum_{l\epsilon L} \sum_{a\epsilon\{A,B,C\}} \left((IL_l^T + CL_l^T) - (IL_{la}^E + CL_{la}^E) - (IN_{la}^{DE} + CN_{la}^{DE}) \right) \cdot X_{la} - \sum_{l\epsilon L} \sum_{n\epsilon\{C,S\}} \sum_{a\epsilon\{B,C\}} K_a^n \cdot (IN_{la}^{nE} + CN_{la}^{nE}) \cdot X_{la}$$

Vincoli (*eq. 3.2*):

$$tol \cdot I^M \leq \sum_{l\epsilon L} \sum_{a\epsilon\{A,B,C\}} (IL_{la}^E + IN_{la}^{DE}) \cdot X_{al} + \sum_{l\epsilon L} \sum_{n\epsilon\{C,S\}} \sum_{a\epsilon\{B,C\}} K_a^n \cdot IN_{la}^{nE} \cdot X_{al} \leq I^M$$

In cui il significato dei parametri è:

- K_a^n indica la riduzione di costo che possono subire le infrastrutture di ricarica per gli autobus di una architettura *a* ad un nodo *n*, dovuto al fatto che più linee possono condividere le infrastrutture di ricarica. In questo modo si possono abbassare i costi operativi e di investimento relativi ai nodi capolinea e fermata.
- X_{al} indica se nella sottorete che si sta considerando è presente la linea *l*, e se si è scelto di elettrificarla con l'architettura *a*. Vale 1 se è presente tale configurazione, 0 altrimenti.

3.2 Lo spazio delle soluzioni

Ogni soluzione di questo problema di ottimizzazione è individuata da:

- Una sottorete *S*, ossia un sottoinsieme di linee di autobus contenuto nell'insieme di *L* linee della rete completa.
- Le architetture con cui si è deciso di elettrificare ogni linea della sottorete *S*.

Indicando con *L* il numero di linee della rete complessiva che si sta analizzando, e sapendo che per questo anno di lavoro si analizzeranno *2 architetture possibili* con cui elettrificare ogni linea (l'architettura *A* e la *B*), si ottiene che lo spazio possibile delle soluzioni è pari a:

$$\sum_{k=1}^{L} \binom{L}{k} \cdot 2^k \qquad (eq. 3.3)$$

L'equazione 3.3 mostra la grandezza dello spazio delle soluzioni. Infatti, le soluzioni possibili vanno cercate tra tutte le possibili combinazioni delle linee di autobus e per ognuna di queste combinazioni vanno analizzate tutte le possibili disposizioni con ripetizione delle due architetture.

In questo modo, con le combinazioni delle linee andremo ad esplorare tutte le possibili sottoreti della rete complessiva. Invece, andando ad analizzare tutte le disposizioni con ripetizione delle due architetture si analizzeranno tutte le scelte possibili di come elettrificare le diverse linee di una sottorete.

Ad esempio lo spazio delle soluzioni di una rete con 100 linee è pari a $5.153 * 10^{47}$. Questo numero ci fa capire come il voler effettuare una ricerca esaustiva della soluzione esatta di questo problema, sia un problema np-hard, la cui complessità cresce esponenzialmente con il numero di linee della rete da analizzare.

3.3 Descrizione della rete di input usata come dataset di input

Ogni caso di studio è composto da due file:

- *linee_output_best.csv*: In questo file sono presenti tutte le linee e per ogni linea tutte le voci di costo relative ad ogni possibile architettura. In particolare si ha una struttura del tipo *(id linea, id architettura, voci di costo)*. In cui le voci di costo si differenziano in:
    - *invest_iniziale_bus*: Costo di investimento (di linea) iniziale per gli autobus.
    - *costo_energia*: Costo operativo di linea per l'energia.
    - *manut_bus*: Costo operativo di linea per la manutenzione degli autobus.
    - *invest_impianti_dep*: Costo d'investimento degli impianti ai depositi.
    - *invest_impianti_cap*: Costo d'investimento degli impianti ai capolinea.
    - *invest_impianti_ferm*: Costo d'investimento degli impianti alle fermate.
    - *manut_impianti_dep*: Costo operativo per la manutenzione degli impianti ai depositi.
    - *manut_impianti_cap*: Costo operativo per la manutenzione degli impianti ai capolinea.
    - *manut_impianti_ferm*: Costo operativo per la manutenzione degli impianti alle fermate.
    - *costo_allacci_dep*: Costo operativo per gli allacci ai depositi.
    - *costo_allacci_cap*: Costo operativo per gli allacci ai capolinea.
    - *costo_allacci_ferm*: Costo operativo per gli allacci alle fermate.
    - *costo_potenza_dep*: Costo operativo per la potenza erogata ai depositi.
    - *costo_potenza_cap*: Costo operativo per la potenza erogata ai capolinea.
    - *costo_potenza_ferm*: Costo operativo per la potenza erogata alle fermate.

- *percorsi.csv*: In questo file sono presenti tutti i percorsi effettuati dalla rete che si sta analizzando e in particolare viene specificato:
    - *percorso_id*: L'id del percorso considerato
    - *linea_id*: L'id della linea a cui si riferisce quel percorso.
    - *capolinea_i*: Il capolinea di partenza del percorso.
    - *capolinea_f*: Il capolinea di arrivo del percorso.
    - *ora_arrivo*: La fascia oraria che si sta considerando.
    - *frequenza*: Il numero di corse effettuate in quella fascia oraria.

Il file *linee_output_best.csv* contiene tutte le voci di costo con cui sono stati calcolati i valori presenti nella funzione obiettivo e nei vincoli, con l'eccezione del valore K_a^n che viene calcolato in base ai percorsi che una linea percorre.

In particolare si ha che:

- $IL_{la}^E = invest_iniziale_bus$
- $CL_{la}^E = costo_energia + manut_bus$
- $IN_{la}^{DE} = invest_impianti_dep$
- $CN_{la}^{DE} = manut_impianti_dep + costo_allacci_dep + costo_potenza_dep$
- $IN_{la}^{CE} = invest_impianti_cap$
- $CN_{la}^{CE} = manut_impianti_cap + costo_allacci_cap + costo_potenza_cap$
- $IN_{la}^{FE} = invest_impianti_ferm$
- $CN_{la}^{FE} = manut_impianti_ferm + costo_allacci_ferm + costo_potenza_ferm$

4 Progettazione di algoritmi paralleli per l'esplorazione esaustiva dello spazio di soluzioni

In questo capitolo si enunceranno i principali risultati a cui si è giunti relativamente alla progettazione degli algoritmi per l'esplorazione esaustiva dello spazio di soluzioni, mostrando i diversi ragionamenti che hanno portato all'applicazione finale. In particolare si analizzerà l'algoritmo ideato per esplorare la rete in modo parallelo ed efficiente, per poi spiegare gli algoritmi di bilanciamento del carico.

4.1 Algoritmo di esplorazione della rete

Per risolvere un problema di ricerca esaustiva, su istanze reali e di dimensioni significative si ha bisogno di calcolo parallelo. Infatti, il punto centrale di questo lavoro ha riguardato proprio la creazione e l'implementazione di un algoritmo di esplorazione della rete in grado di essere parallelizzato. Prima di analizzare gli algoritmi di bilanciamento del carico che si sono implementati si analizzerà il modo in cui l'insieme di linee possono essere esplorate.

Il problema di ottimizzazione che si vuole risolvere deve essere in grado di analizzare tutte le possibili combinazioni delle linee di autobus e per ognuna di queste combinazioni, si devono valutare tutte le possibili disposizioni con ripetizione delle due architetture analizzate per questo anno.

Tutti i ragionamenti dietro ogni algoritmo che verrà spiegato e analizzato sono nati da una semplice idea, ossia trovare un algoritmo in grado di:

- Far esplorare lo spazio delle soluzioni in modo parallelo.
- Far esplorare ogni possibile soluzione una e una sola volta e ad uno e un solo processore.

Immaginiamo di avere una rete complessiva formata da 5 linee come la seguente:

{0, 1, 2, 3, 4} (Esempio 4.1)

Tale rete è composta dalle linee di autobus numero 0, 1, 2, 3 e 4. Dovendo suddividere il calcolo relativo ad un insieme di linee, si è ragionato sulla possibilità di esplorare per ogni linea *x* tutte le sottoreti contenenti *x*.

Tuttavia, in questo caso ci sarebbe una sovrapposizione nelle soluzioni esplorate, poiché tra le combinazioni di sottoreti contenenti la linea *y* e quelle contenenti la linea *x* (con *x* diverso da *y*), si andrebbero ad analizzare due volte le sottoreti contenenti sia *x* che *y*. Facendo riferimento all'esempio di cinque linee, quando si calcolerebbero le combinazioni contenenti la linea 1 si andrebbero ad esplorare anche la sottorete {1, 2}, ma anche quando si esplorerebbero le combinazioni contenenti la linea 2 si analizzerebbe quella stessa sottorete.

Per tale motivo, si è seguito un approccio diverso, in grado di far calcolare ogni possibile soluzione solo una volta. Riprendendo l'esempio con cinque linee, si nota come le linee possono essere indicizzate, dando loro un ordine ben preciso. La linea 0 in posizione 0, la 1 in posizione 1 e così via.

L'esplorazione implementata fa in modo tale di far calcolare tutte le possibili combinazioni di una linea x, con le linee che si trovano in una posizione più alta rispetto a quella occupata da x. In tal modo nel nostro esempio si otterrà che:

- Esplorare la linea 0, significherà analizzare tutte le sottoreti contenenti la linea 0. Ossia le sottoreti: {0} , {0, 1} , {0, 2} , {0, 3} , {0, 4} , {0, 1, 2} , {0, 1, 3} , {0, 1, 4} , {0, 2, 3} , {0, 2, 4} , {0, 3, 4} , {0, 1, 2, 3} , {0, 1, 2, 4} , {0, 1, 3, 4} , {0, 2, 3, 4} , {0, 1, 2, 3, 4}. In particolare significherà analizzare tutte le reti di colore giallo presenti in figura 4.1.

- Esplorare la linea 1, significherà analizzare tutte le sottoreti contenenti almeno la linea 1, ma che non contengono la linea 0. Ossia le sottoreti: {1} , {1, 2} , {1. 3} , {1, 4} , {1, 2, 3} , {1, 2, 4} , {1, 3, 4} , {1, 2, 3, 4}. Vedi reti di colore verde in figura 4.1.
- Esplorare la linea 2, significherà analizzare tutte le sottoreti contenenti almeno 2, ma che non contengono 0 e 1. Ossia le sottoreti: {2} , {2, 3} , {2, 4} , {2, 3, 4}. Vedi reti di colore celeste in figura 4.1.
- Esplorare la linea 3 significherà analizzare tutte le sottoreti contenenti almeno 3, ma che non contengono 0, 1 e 2. Ossia le sottoreti: {3} , {3, 4}. Vedi reti di colore arancione in figura 4.1.
- Esplorare la linea 4 significherà analizzare tutte le sottoreti contenenti almeno 4, ma che non contengono 0, 1, 2 e 3. Ossia le sottoreti: {4}. Vedi reti di colore bianco in figura 4.1.

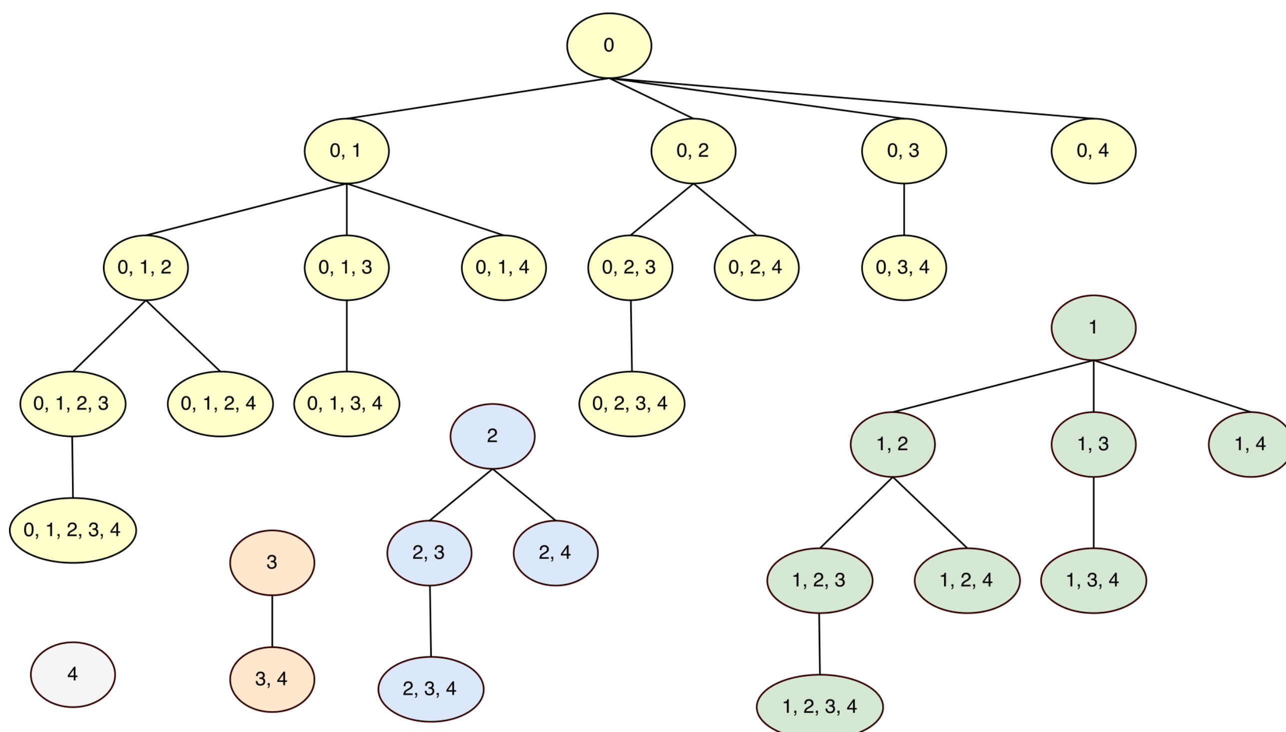

Figura 4.1: Alberi che rappresentano il modo in cui si può esplorare una linea di autobus. In giallo tutte le sottoreti che si esplorano partendo dall'analisi della linea 0, in grigio quelle che si esplorano analizzando la linea 1 e così via.

Quello appena descritto è un approccio di come si può pensare di suddividere il calcolo. Nella prossima sottosezione si cercherà di spiegare come si è implementata tale esplorazione.

4.1.1 Metodo di Branch and bound

Si prenda in esame la rete nell'esempio 4.1, e si consideri di esplorare le combinazioni della linea 0.

La figura 4.2 mostra come si potrebbe pensare di effettuare il branch delle soluzioni. Partendo dalla possibile soluzione della singola linea e procedendo fino ad arrivare ad esplorare tutte le combinazioni. Inoltre, si può notare come i nodi figli nell'albero, altro non sono che delle sottoreti più grandi di 1 linea rispetto al nodo padre.
Questo ci permette di effettuare il taglio delle soluzioni. Infatti, se una soluzione padre eccede il budget (per tutte le possibili architetture di ricarica), sicuramente la soluzione figlia eccederà il budget dedicato agli investimenti. In questo modo, in caso di budget non molto grande si riesce ad evitare di calcolare una grande parte dello spazio delle soluzioni.

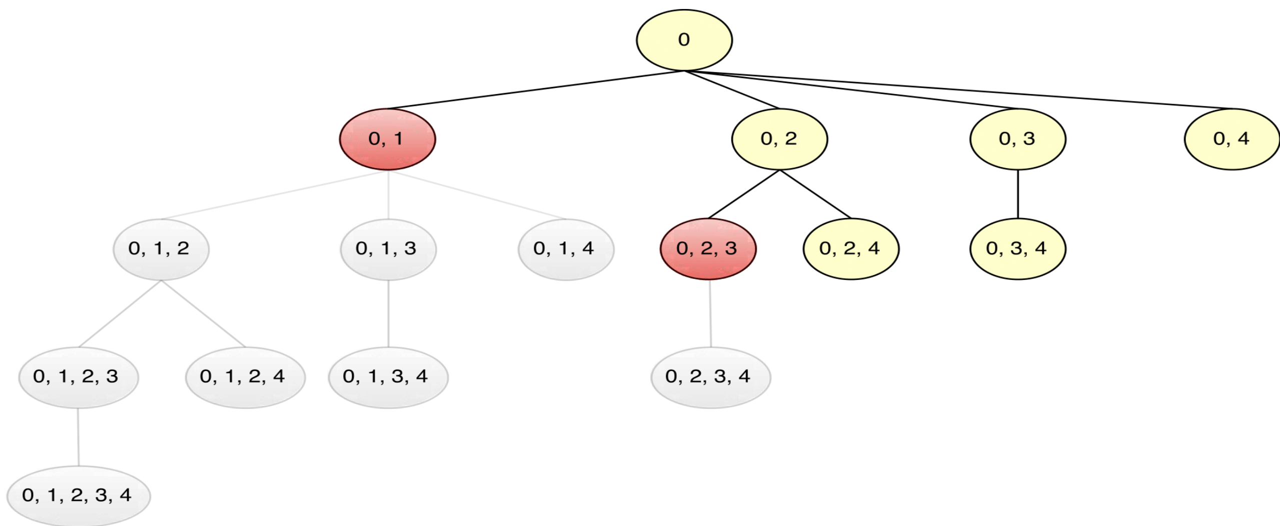

Figura 4.2: Albero in cui viene mostrato un esempio di applicazione possibile della tecnica di branch and bound. I nodi in rosso rappresentano le sottoreti il cui costo di investimento ha superato il budget a disposizione. I nodi in bianco rappresentano le reti che sono state tagliate dall'esplorazione in quanto il loro costo di investimento avrebbe superato il budget a disposizione. I nodi in giallo rappresentano le reti analizzate e per cui c'è almeno una combinazione di architetture il cui costo di investimento è inferiore al budget massimo di investimento.

Algorithm 4.1 Esplorazione della rete con tecnica di branch e bound

```
 1:  procedure exploreNetwork(lowerIndex, upperIndex, pos, chosenLines[ ])
 2:      if lowerindex ≥ numBusLines then
 3:          return
 4:      end if
 5:      i = lowerIndex
 6:      while i < numBusLines && !(pos == 0 && i ≥  upperIndex) do
 7:          chosenLines[pos] = i
 8:          if analyseAllArchitectures(chooseLines) then
 9:              exploreNetwork(i+1, upperIndex, pos+1, chosenLines)
10:          end if
11:          i + +
12:      end while
13:  end procedure
```

Come si vede dall'algoritmo 4.1 l'esplorazione avviene in maniera ricorsiva, grazie ad un array che mantiene gli indici delle linee da analizzare di volta in volta. Il taglio dello spazio delle soluzioni avviene nel momento in cui dopo aver analizzato tutte le disposizioni delle architetture per una sottorete, ci si rende conto che quella sottorete eccede sempre il budget. In tal caso si fermerà la ricorsione, e non si chiamerà nuovamente la exploreNetwork.
È da sottolineare l'importanza che hanno i quattro parametri della funzione *exploreNetwork*. Infatti, cambiando i parametri che si passano alla funzione si è in grado di esplorare una porzione della rete, piuttosto che un'altra. L'analisi dei parametri da passare alla funzione *exploreNetwork* per esplorare una sezione della rete è alla base di alcuni degli algoritmi di bilanciamento implementati. Prima di approfondire i diversi algoritmi si analizzerà il protocollo di comunicazione tra master e slave.

4.1.2 Comunicazione tra master e slaves

Il calcolo da effettuare viene suddiviso dal nodo master che, eseguendo un determinato algoritmo di bilanciamento, comunicherà ad ogni nodo slave il carico di lavoro da svolgere. Astraendo dall'algoritmo di load balancing implementato si mostrerà il formato dei messaggi scambiati tra master e slaves.
Uno slave riceverà dal nodo master un array contenente numeri interi. La posizione di tali numeri segue un formato ben preciso. Il protocollo di comunicazione ha il seguente schema:

$$[LI, UI, POS, \underbrace{\ldots}_{POS\ elements}, LI, UI, POS, \underbrace{\ldots}_{POS\ elements}, LI, UI, POS, \underbrace{\ldots}_{POS\ elements}]$$

Laddove:

- *LI* è il lower index.
- *UI* è l'upper index.
- *POS* è la posizione da riempire nell'array chosenLines.
- … sono i numeri che rappresentano le posizioni delle linee scelte per la sottorete che si sta vuole analizzare.

Si cercherà di spiegare il protocollo tramite un esempio:

[3, 4, 0, 3, 1, 3, 0, 1, 2, 0, 3, -1] (Esempio 4.2)

In questo caso lo slave che riceve questo array, sa esattamente quali porzioni di rete deve esplorare, in particolare si ha:

$$[\overbrace{\underbrace{3}_{LI}, \underbrace{4}_{UI}, \underbrace{0}_{POS}}^{prima\ porzione\ di\ rete}, \overbrace{\underbrace{3}_{LI}, \underbrace{1}_{UI}, \underbrace{3}_{POS}, \underbrace{0,\ 1,\ 2}_{chosenLines[\ ]}}^{seconda\ porzione\ di\ rete}, \overbrace{\underbrace{0}_{LI}, \underbrace{3}_{degree}, \underbrace{-1}_{special\ character}}^{terza\ porzione\ di\ rete}]$$

Alla ricezione di questo array, il processore esplorerà le seguenti porzioni di rete:

- La prima porzione di rete è quella che viene esplorata chiamando la funzione *exploreNetwork(3, 4, 0, [])*. In tal modo saranno analizzate tutte le combinazioni di sottoreti con almeno la linea in posizione 3, senza le linee in posizione 0, 1 e 2. Quindi in questo caso verrà esplorata la porzione di rete formata dalle reti: {3} , {3, 4}. (Cfr. reti arancione presenti in figura 4.3).
- La seconda porzione di rete è quella che viene esplorata chiamando la funzione *exploreNetwork(3, 1, 3, [0, 1, 2])*. In questo caso verranno esplorate le sottoreti che hanno sia la linea 0, sia la 1 e anche la 2, esclusa la rete {0, 1, 2}. In particolare si esploreranno le seguenti sottoreti: {0, 1, 2, 3} , {0, 1, 2, 4} , {0, 1, 2, 3, 4}. (Cfr. reti blu presenti in figura 4.3).
- La terza porzione di rete è più particolare delle altre. I tre valori che solitamente hanno significato di lowerIndex, upperIndex e pos, in questo caso assumono un significato diverso. Il modo per capire che ci si trova in questa situazione è il carattere speciale -1 che indica che si vuole esplorare la rete in un altro modo.
  In effetti in questo caso si andranno ad esplorare tutte le possibili combinazioni con grandezza massima fino a *degree*, delle reti contenenti almeno la linea in posizione *LI*. Nell'esempio riportato la porzione di rete che si andrà ad esplorare è: {0} , {0, 1} , {0, 2} , {0, 3}, {0, 4} , {0, 1, 2} , {0, 1, 3} , {0, 1, 4} , {0, 2, 3} , {0, 2, 4} , {0, 3, 4}. (Cfr. reti rosse presenti in figura 4.3).

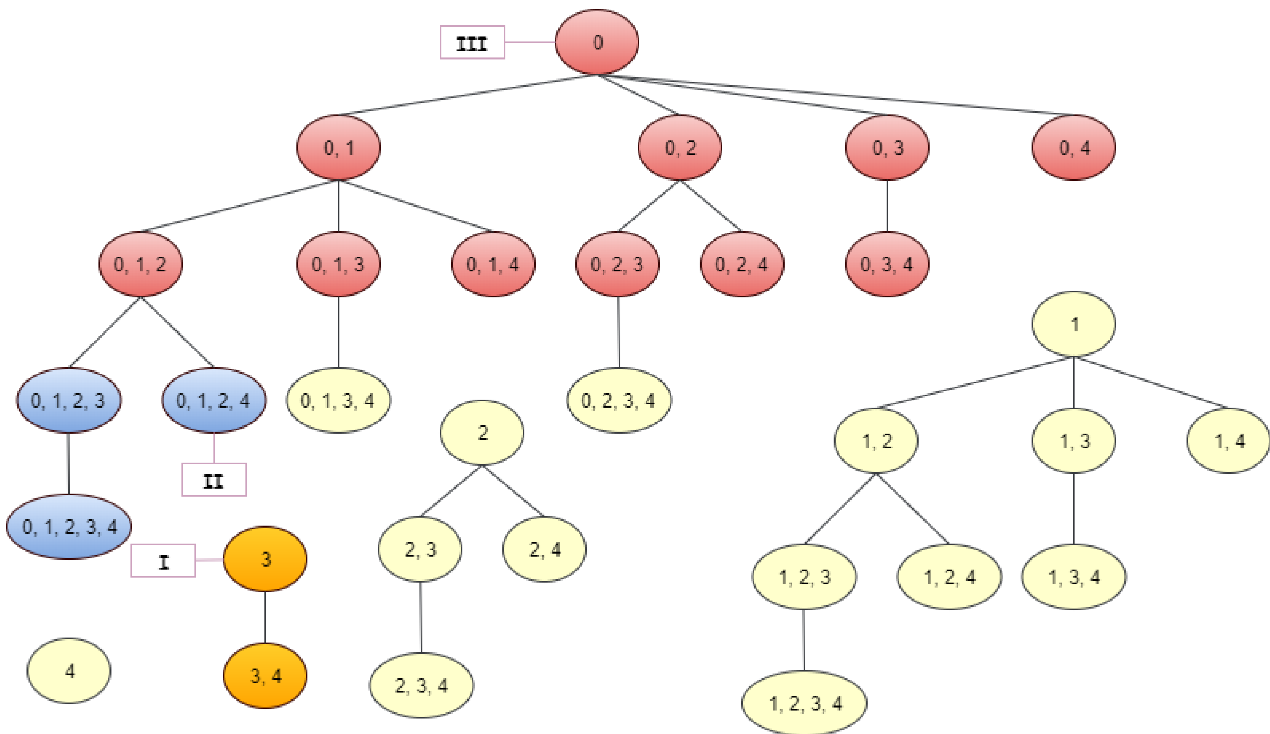

Figura 4.3: In questa figura vengono mostrate le diverse porzioni di rete che vengono esplorate dal processore che riceve l'array di interi riportato nell'esempio 4.2. Tali porzioni sono quelle colorate in arancione, rosso e blu.

Avendo compreso il modo in cui esplorare la rete, e il formato dei dati scambiati, si può passare ad analizzare i diversi algoritmi di load balancing implementati per questa applicazione.

4.2 Algoritmi di bilanciamento

4.2.1 Bilanciamento di linee equo

Il primo algoritmo di bilanciamento che è stato implementato suddivide il calcolo in base alle linee e lo fa assegnando un numero equo di linee da analizzare ad ogni processore. Come visto in precedenza, il carico computazionale delle linee non è lo stesso, ma dipende dalla posizione che tali linee occupano nel momento in cui viene fatta l'inizializzazione della struttura dati che le conterrà in fase di esecuzione.
Più precisamente il numero di sottoreti assegnate sarà maggiore per i processori a cui verranno assegnate le linee nelle prime posizioni, e molto minore per i processori che analizzeranno le linee nelle ultime posizioni.

Algorithm 4.2 Load balancing delle linee in modo equo

```
1: procedure balanceEqualMode(numLines, numProc)
2:     if numProc > numBusLines then
3:         associa una linea ad un processore
4:     else
5:         associa numBusLines/numProc ad ogni processore
6:         ripartisci eventuali linee in eccesso agli ultimi processori
7:     end if
8: end procedure
```

Per comprendere vediamo come si comporterebbe l'algoritmo nel nostro esempio 4.1. Immaginando di avere 4 processori su cui eseguire il calcolo, l'algoritmo andrebbe ad assegnare:

- Al primo processore il calcolo relativo alla linea 0. Ossia, come analizzato in precedenza verrebbero assegnate 2^4 sottoreti da analizzare. Inoltre, tra le 16 sottoreti ci sono le sottoreti più grandi, il che

in assenza di taglio implica un calcolo molto oneroso rispetto a quello degli altri processori. (cfr. nodi color giallo in figura 4.4).

- Al secondo processore il calcolo relativo alla linea 1. Ossia 2^3 sottoreti da analizzare. (cfr. nodi color arancione in figura 4.4).
- Al terzo processore il calcolo relativo alla linea 2. Ossia 2^2 sottoreti. (cfr. nodi color rosso in figura 4.4).
- Al quarto processore il calcolo relativo alla linea 3 e 4. Ossia $2^1 + 2^0$ sottoreti. (cfr. nodi color celeste in figura 4.4).

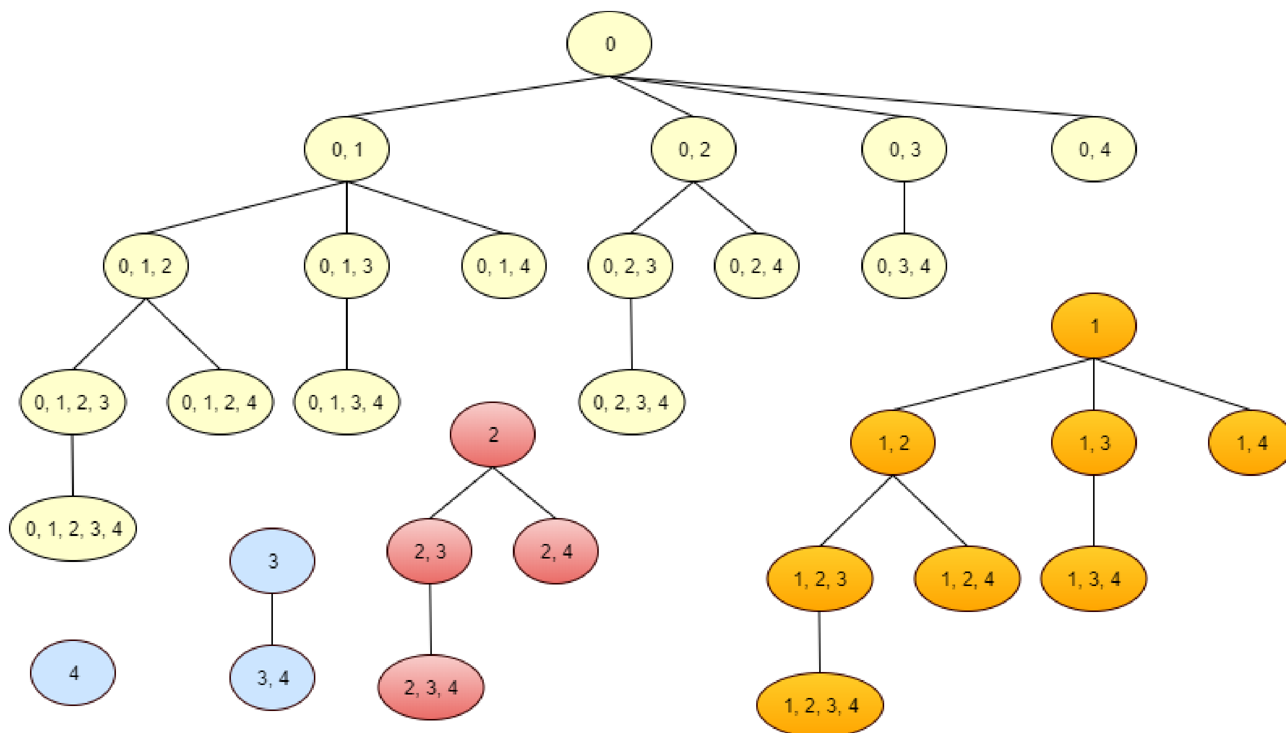

Figura 4.4: In questa figura viene riportato l'esempio di come si comporta l'algoritmo di bilanciamento di linee equo nel caso di 5 linee e 4 processori.

Questo algoritmo ha molti svantaggi, a fronte di un solo vantaggio che è la facilità di progettazione e implementazione. I limiti di questo algoritmo sono molteplici. Ad esempio il numero massimo di processori che si riesce a sfruttare è pari al numero di linee della rete analizzata. Se abbiamo più processori che linee, ci sarebbero dei processori che non riuscirebbero ad effettuare alcun tipo di lavoro. Inoltre, il carico è sbilanciato sia per numero di sottoreti analizzate dai singoli processori, sia per la grandezza di tali sottoreti.

Tuttavia, va tenuto conto che il numero di sottoreti assegnate non corrisponde al vero numero di sottoreti che poi nella realtà verranno analizzate. Questo perché in base al valore del budget, l'algoritmo di esplorazione riesce a tagliare il numero di soluzioni da esplorare.
Per questo motivo si può parlare di due diverse metriche:

- Numero di sottoreti assegnate teoricamente. Si tratta del massimo numero di sottoreti da esplorare, che coincide con il numero di reti analizzate solo quando l'algoritmo non riesce ad effettuare dei tagli.
- Numero di sottoreti analizzate realmente. Questo numero si allontana tanto più dal numero di sottoreti assegnate, tanto quanto il budget è basso e non permette di elettrificare sottoreti molto grandi.

4.2.2 Bilanciamento di linee first less, last more

Questo algoritmo è un raffinamento del primo che può portare delle migliorie al load balancing visto nella precedente sezione, ma solo nel caso in cui il numero di linee della rete da esplorare è maggiore del numero di processori.
Lo svantaggio principale del bilanciamento di linee equo è che il processore che deve analizzare la linea in posizione 0, effettua un calcolo molto più grande, rispetto agli altri processori. Sulla base di tale premessa, questo algoritmo cerca di agire proprio sul problema. L'idea è di suddividere i processori in gruppi, e di assegnare il calcolo di una sola linea ad ogni singolo processore appartenente al primo gruppo, mentre ad ogni processore del secondo gruppo si assegneranno 2 linee, e così via, fino a ripartire le rimanenti linee tra i processori dell'ultimo gruppo. Infatti, le linee che si assegneranno ai primi processori sono le più onerose dal punto di vista del calcolo da effettuare, in quanto sono quelle nelle prime posizioni.

Come viene espresso dal nome dell'algoritmo, si assegnano meno linee ai primi processori, e più linee agli ultimi. Questo perché le linee che vengono assegnate ai primi processori hanno un carico di lavoro maggiore rispetto alle altre.

Per comprendere appieno l'algoritmo ne viene riportato lo pseudocodice:

Algorithm 4.3 Load balancing delle linee in rst less, last more

```
 1: procedure balanceFirstLessLastMore(numLines, numProc)
 2:     quantum = numBusLines / numProc
 3:     if quantum < numProc then
 4:         granularity = numProc / quantum
 5:         numGroups = quantum
 6:     else
 7:         granularity = 1
 8:         numGroups = numProc
 9:     end if
10:     i = 1
11:     while i < numGroups do
12:         assegna i linee ad ogni proc. ∈ all'i-esimo gruppo
13:     end while
14:     ripartisci le linee rimaste tra i processori dell'ultimo gruppo
15: end procedure
```

Il vantaggio di questo algoritmo è quello di aver introdotto dei miglioramenti rispetto alla versione base, che sono stati in effetti un ottimo spunto per poter progettare algoritmi maggiormente sofisticati.
Tuttavia, gli svantaggi visti per il primo algoritmo rimangono anche in questo caso. Il carico risulta essere sbilanciato e non si riescono a sfruttare appieno le risorse di calcolo, perché la distribuzione del carico può avvenire al massimo su di un numero di processori pari al numero di linee della rete analizzata.

Di seguito (fig. 4.5) è riportato il caso in cui il carico dell'esempio di 5 linee viene bilanciato con l'algoritmo di *firstLessLastMore* (caso con 2 processori). In tal caso si ha un notevole beneficio in quanto il numero di sottoreti assegnate ai processori è ben bilanciato.

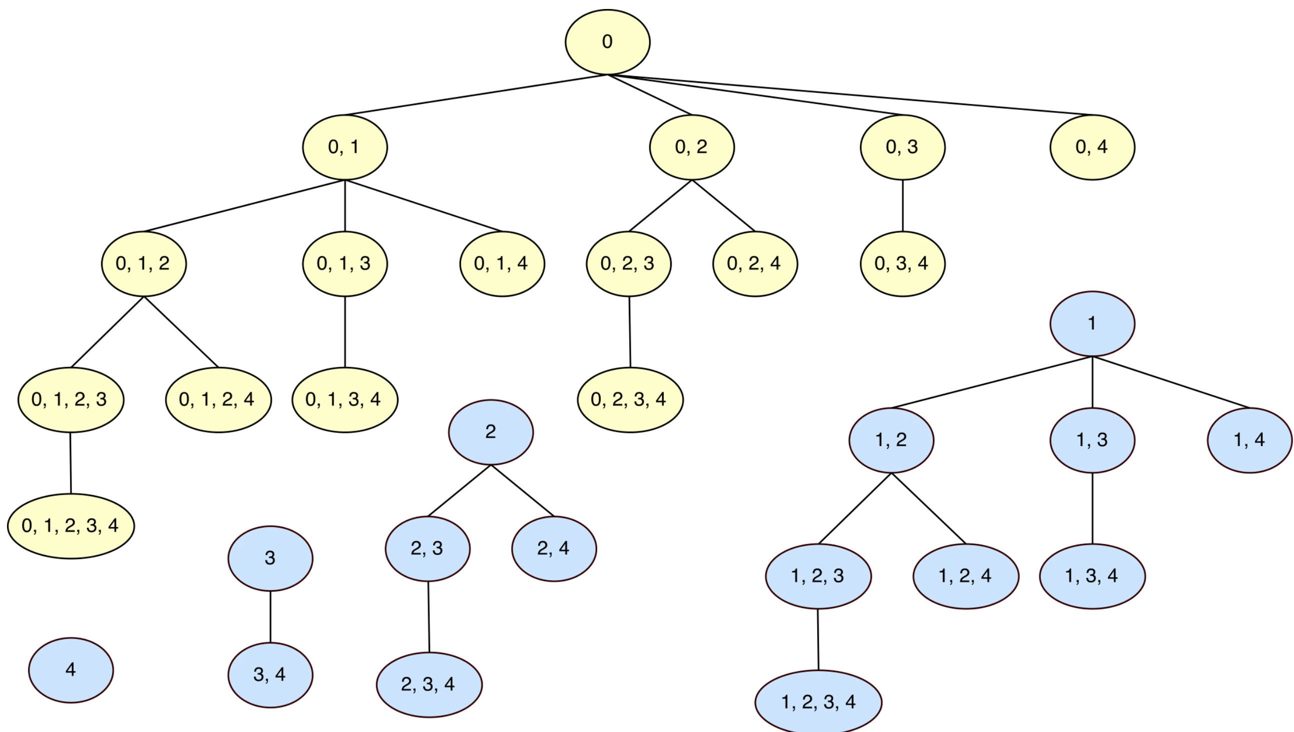

Figura 4.5: In questa figura viene riportato l'esempio di come si comporta l'algoritmo di bilanciamento firstLessLastMore nel caso di 5 linee e 2 processori.

4.2.3 Bilanciamento fisso in profondità

I bilanciamenti visti finora hanno un grande limite, ossia l'incapacità di poter suddividere il calcolo oltre un certo numero di processori. Il load balancing fisso in profondità permette di suddividere porzioni più piccole di rete tra i diversi nodi di calcolo. Infatti, in questo caso la suddivisione dello spazio di soluzioni non è legata solamente alla singola linea, ma anche alle coppie o n-uple di linee.

L'idea è quella di poter affidare ad un processore il calcolo di una porzione di rete più piccola, come ad esempio tutte le possibili sottoreti che hanno sia la linea in posizione 0 che in posizione 1, oppure tutte le sottoreti che hanno sia la linea 0, 2 e 3. In questo modo si riesce sia a sfruttare più processori, qualora presenti, ma anche a suddividere più in profondità il carico di lavoro.

Per comprendere tale algoritmo viene riportato di seguito lo pseudocodice:

```
Algorithm 4.4 Load balancing fisso in profondità
1:  procedure balanceDeepMode(procToDivide, degree)
2:      ∀ processore la cui computazione non si deve dividere
3:          assegna le linee che avrebbe attribuito firstLessLastMore
4:      ∀ processore la cui computazione si deve dividere
5:          ∀ linea che che firstLessLastMore avrebbe assegnato a quel processore
6:              applica la suddivisione della computazione in base a degree
7:              assegna le porzioni di rete in modo round robin ai processori
8:  end procedure
```

Come si può notare, l'algoritmo si basa su due parametri fondamentali: *degree* e *numProcToDivide*. Il *degree* indica il grado di suddivisione del carico. Mentre *numProcToDivide* indica il numero di processori il cui carico che gli avrebbe assegnato l'algoritmo *firstLessLastMore* è da dividere.

Vediamo come si comporterebbe l'algoritmo nel caso dell'esempio 4.1 con *degree* 2 e numero di processori da dividere solo i primi due (dei quattro complessivi):

- Il terzo e il quarto processore non devono suddividere la porzione dello spazio di soluzioni che l'algoritmo *firstLessLastMore* gli avrebbe assegnato. Dunque vengono assegnate al terzo processore il calcolo della linea 2 e al quarto processore l'esplorazione delle linee 3 e 4.
- *degree* = 2, primo processore: al primo processore l'algoritmo *firstLessLastMore* avrebbe assegnato l'esplorazione della linea 0. Ma tale processore è tra quelli il cui carico deve essere diviso in base a *degree* = 2 (ossia in base alle coppie). Dunque, si assegnerà l'esplorazione di tutte le combinazioni di sottoreti contenenti {0, 1} ad un processore, e poi con modalità round robin (dando precedenza ai processori a cui non è stata ancora assegnata alcun lavoro) si assegneranno {0, 2} , {0, 3}. Invece, {0, 4} non viene assegnata perché è un nodo senza figli.
- *degree* = 2, secondo processore: al secondo processore l'algoritmo *firstLessLastMore* avrebbe assegnato l'esplorazione della linea 1. Ma tale processore è tra quelli il cui carico deve essere diviso in base a *degree* = 2 (ossia in base alle coppie). Dunque, si assegnerà con modalità round robin l'esplorazione di tutte le combinazioni di sottoreti contenenti {1, 2} , {1, 3}.

In questo modo si sarà ottenuto un bilanciamento del tipo:

- Primo processore: Dovrà esplorare al massimo 10 sottoreti. Infatti, si occuperà di tutte le possibili combinazioni di linee con {0, 1} e {1, 2}.
- Secondo processore: Dovrà esplorare al massimo 4 sottoreti. Infatti, si occuperà di tutte le possibili combinazioni di linee con {0, 2}, e {1, 3}.
- Terzo processore: Dovrà esplorare al massimo 9 sottoreti. Infatti, si occuperà di tutte le possibili combinazioni di linee con {2} e {0, 3} e tutte le sottoreti contenenti {1} grandi al massimo 2 (e non contenenti una linea in posizione precedente alla posizione 1).
- Quarto processore: Dovrà esplorare al massimo 8 sottoreti. Infatti, si occuperà di tutte le possibili combinazioni di linee con {3}, {4} e tutte le sottoreti contenenti {0} grandi al massimo 2.

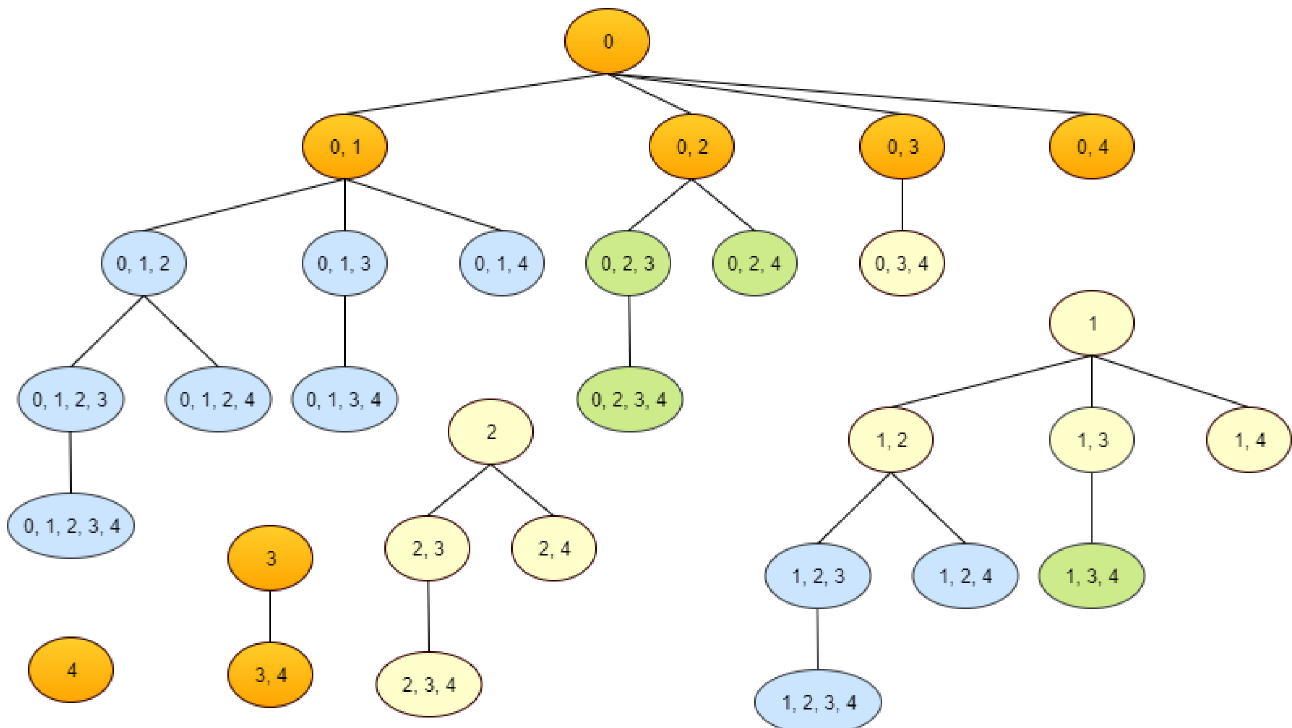

Figura 4.6: In questa figura viene riportato l'esempio di come si comporta l'algoritmo di bilanciamento fisso in profondità nel caso di 5 linee, 4 processori, degree pari a 2 e numero di processori il cui iniziale assegnamento del carico deve essere suddiviso è 2.

Implementazione dell'assegnamento di n-uple

Si è descritto l'algoritmo, ma non si è mostrato come sia possibile assegnare le combinazioni delle sottoreti in base alle coppie o alle n-uple.

L'idea che sta dietro questa implementazione è legata all'algoritmo ricorsivo di esplorazione della rete. Ad ogni chiamata della funzione *exploreNetwork()* corrispondono determinati parametri passati alla funzione, ovvero *lowerIndex*, *upperIndex*, *pos* e *chosenLines[ ]*.

Per capire meglio, di seguito viene mostrato (in figura 4.7) l'albero dei parametri passati alla funzione per poter attivare la ricorsione dell'esplorazione.

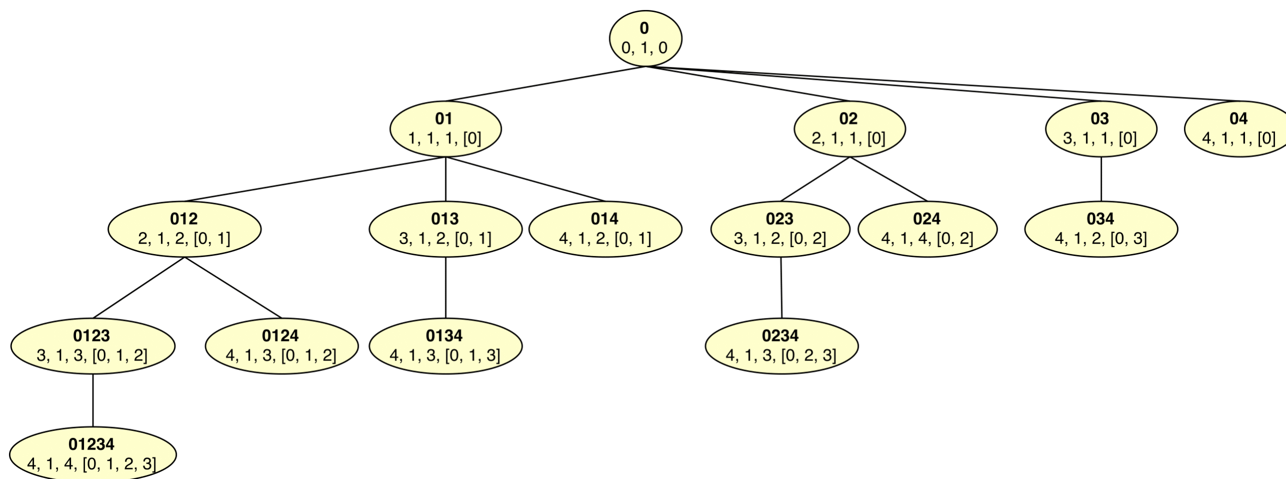

Figura 4.7: Albero che rappresenta i parametri passati alla funzione exploreNetwork per calcolare le combinazioni delle linee che contengono almeno la linea 0, in una rete che contiene 5 linee.

In particolare si ha che chiamando la funzione di *exploreNetwork* con i valori contenuti in un nodo di tale albero si aziona l'esplorazione di tutte le sottoreti dei propri nodi discendenti, dei nodi fratelli e dei discendenti dei fratelli.
Ad esempio se si chiama la *exploreNetwork* con i parametri (2, 1, 2, [0, 1]), si andrà ad esplorare:

- La sottorete corrispondente al nodo che ha avviato la ricorsione: {0, 1, 2}.
- Tutti i propri nodi figli, ossia le sottoreti: {0, 1, 2, 3}, {0, 1, 2, 4}, {0, 1, 2, 3, 4}.
- Tutti i nodi fratelli, ossia le sottoreti: {0, 1, 3}, {0, 1, 4}.
- Tutti i discendenti dei nodi fratelli: {0, 1, 3, 4}.

4.2.4 Bilanciamento variabile in profondità

Il bilanciamento variabile in profondità è un raffinamento del precedente algoritmo. Come analizzato in precedenza, il calcolo maggiore è legato alle prime linee. Basti pensare che:

$$\textit{Sottoreti assegnate alla i-esima linea} = 2^{numBusLines-1-i} \qquad \text{(Formula 4.1)}$$

Per tale motivo si è pensato di poter effettuare una suddivisione del carico variabile: associare un *degree* alto alle prime linee, e far diminuire il *degree* assegnato man mano che le linee hanno associato un carico di lavoro minore.

Per comprendere appieno l'algoritmo ne viene riportato lo pseudocodice:

Algorithm 4.5 Load balancing con profondità variabile

1: procedure *balanceDynamicDeepMode*(procToDivide, maxDegree)
2: dividi i processori in *maxDegree-1* gruppi
3: i = 0
4: while i < maxDegree - 1 do
5: dividi il calcolo delle linee del gruppo i con grado *maxDegree - i*
6: i + +
7: end while
8: end procedure

Questo algoritmo pur avendo un grado minore di profondità per alcune linee, in realtà riesce ad alleggerire l'algoritmo di bilanciamento, consentendo uno scambio minore di dati tra master e slave in fase di bilanciamento, e ottiene dei risultati anche migliori rispetto al precedente algoritmo.

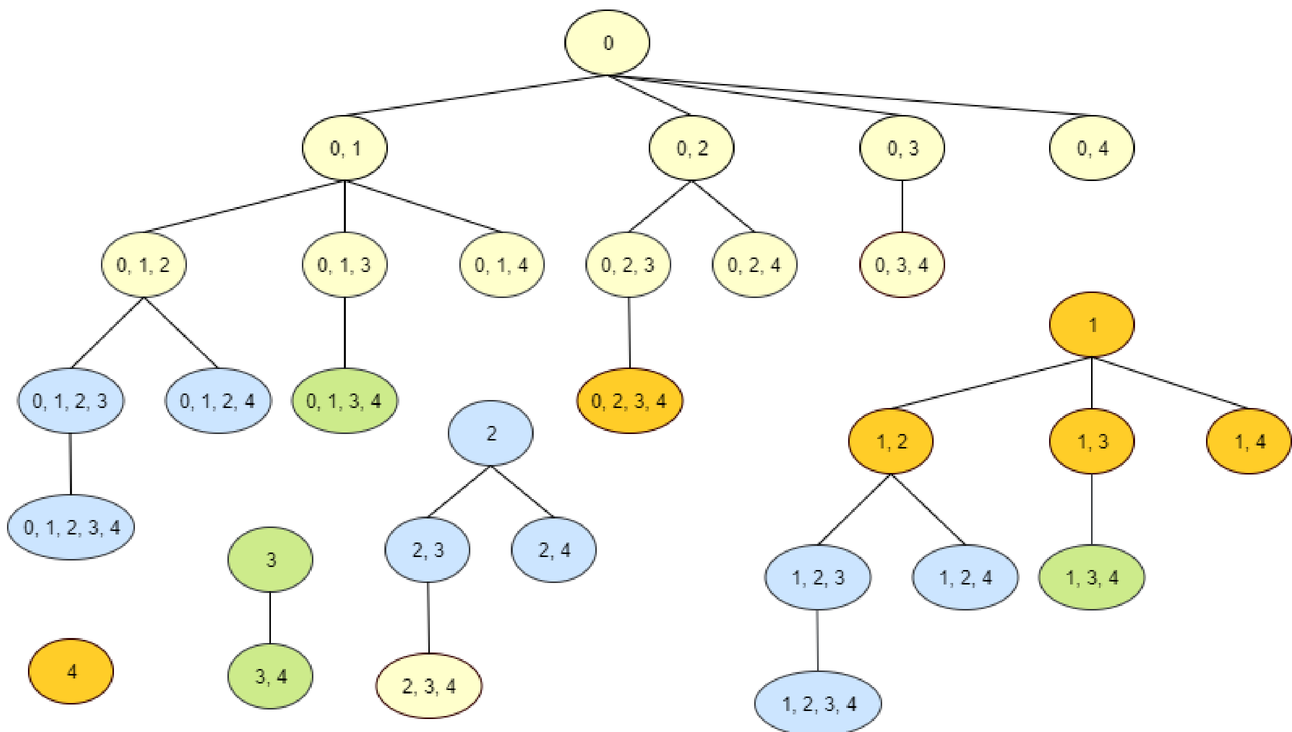

Figura 4.8: In questa figura viene riportato l'esempio di come si comporta l'algoritmo di bilanciamento variabile in profondità nel caso di 5 linee, 4 processori, degree pari a 3 e numero di processori il cui iniziale assegnamento del carico deve essere suddiviso è 4.

5 Tecnologia sviluppata

5.1 Metodologie usate

5.1.1 MPI e C++

Per l'esecuzione dei test del programma realizzato per questo lavoro di ricerca, si sono utilizzate le risorse di calcolo di Cresco 3 e Cresco 4. La scelta per la scrittura del programma è ricaduta sul linguaggio C++ e sul protocollo di comunicazione tra processi MPI [2].

Il motivo di usare il C++ piuttosto che altri linguaggi è legato alle prestazioni offerte dal linguaggio. Inoltre, il C++ offre un mezzo di astrazione che, al contrario di altri linguaggi, non produce overhead delle prestazioni a runtime. In questo modo si può scrivere un codice efficiente, che ha anche un alto livello di astrazione.

Il programma pensato per essere eseguito su molteplici nodi del cluster necessita di un protocollo di comunicazione tra i computer. La scelta in questo caso è ricaduta su MPI (Message Passing Interface). Infatti, MPI è lo standard per la comunicazione tra nodi appartenenti a un cluster di computer che eseguono un programma parallelo sviluppato per sistemi a memoria distribuita.

Message passing Interface è un protocollo il cui consenso è condiviso da più di 40 organizzazioni partecipanti, tra cui vendor, ricercatori, sviluppatori di librerie software e utenti. Tuttavia, MPI non è uno standard IEEE o ISO, ma è diventato di fatto uno standard delle industrie per poter scrivere programmi che hanno bisogno di scambiare messaggi su piattaforme di HPC.

L'obiettivo principale di MPI è quello di stabilire uno standard flessibile, efficiente e portabile. Infatti, MPI rispetto alle precedenti librerie utilizzate per il passaggio di parametri tra nodi, ha il vantaggio di essere molto portabile (è stata implementata per moltissime architetture parallele) e veloce (viene ottimizzata per ogni architettura).

È importante sottolineare il fatto che MPI non è una libreria, ma piuttosto la specifica di come una libreria di message passing dovrebbe essere realizzata. MPI si occupa principalmente del modello di programmazione parallela a passaggio di messaggi: i dati si muovono dallo spazio di indirizzamento di un processo a quello di un altro, attraverso operazioni combinate su ciascun processo. Tutto il parallelismo è esplicito: il programmatore è responsabile dell'identificazione corretta del parallelismo e dell'implementazione di algoritmi paralleli usando i costrutti MPI. Per la compilazione del programma si è usata l'implementazione di MPI che si chiama "Open MPI" (versione 1.4.3).

5.2 Programma con load balancing statico

5.2.1 Struttura del programma

Il programma prevede l'elaborazione delle informazioni contenute in due file che hanno uno specifico formato. Si tratta di:

- Linee_output_best.csv: In questo file sono contenute le informazioni relative ad ogni singola linea in termini di id della linea, architettura, fattibilità e costi.
- Percorsi.csv: In questo file invece sono contenuti le informazioni sui percorsi delle linee specificati per fascia oraria.

Il programma che si è realizzato per il caso applicativo è costituito da quattro fasi principali:
1. Inizializzazione dell'ambiente MPI, delle variabili e degli oggetti necessari per l'analisi della rete.
2. Il bilanciamento del carico, effettuato dal nodo master, per comunicare a tutti i processori il carico di lavoro da svolgere.
3. L'esplorazione della porzione di rete che è stato deciso dall'algoritmo di load balancing.
4. Il merge dei risultati che sono stati calcolati dai diversi processori.

Per poter implementare queste quattro fasi si è realizzato un progetto costituito nel seguente modo:

```
Onebus
|-- static_program
|   |-- mpi
|   |   |-- balancing.cpp
|   |   |-- balancing.hpp
|   |   |-- balancing_help_functions.cpp
|   |   |-- balancing_help_functions.hpp
|   |   |-- merge_results.cpp
|   |   |-- merge_results.hpp
|   |   |-- mpi_functions.cpp
|   |   `-- mpi_functions.hpp
|   |-- variables
|   |   `-- static_variables.cpp
|   `-- onebus_static_main.cpp
`-- libraries
    |-- constants
    |   `-- constants.cpp
    |-- exploration
    |   |-- exploration.cpp
    |   |-- exploration.hpp
    |   |-- objective_function.cpp
    |   `-- objective_function.hpp
    |-- objects
    |   |-- electrical_arch.cpp
    |   |-- electrical_arch.hpp
    |   |-- line.cpp
    |   |-- line.hpp
    |   |-- network.cpp
    |   |-- network.hpp
    |   |-- traditional_arch.cpp
    |   `-- traditional_arch.hpp
    |-- print
    |   |-- print_functions.cpp
    |   |-- print_functions.hpp
    |   |-- write_functions.cpp
    |   `-- write_functions.hpp
    `-- util
        |-- computation_size.cpp
        |-- computation_size.hpp
        |-- file_management.cpp
        |-- file_management.hpp
        |-- initialization.cpp
        |-- initialization.hpp
        |-- list_network.cpp
        |-- list_network.hpp
        |-- parse_arguments.cpp
        |-- parse_arguments.hpp
        |-- split.cpp
        |-- timestamp.cpp
        `-- timestamp.hpp
```

Il progetto è costituito dalle due cartelle principali: `static_program` e `libraries`.
Nella cartella `static_program` vi è:

- Il file `onebus_static_main.cpp` che contiene il `main` del programma
- La cartella `mpi` che `contiene` tutte le funzioni che utilizzano chiamate MPI e in particolare:

  - Le funzioni di bilanciamento dello spazio delle soluzioni (file `balancing` e `balancing_help_functions`).
  - Le funzioni che permettono di effettuare il merge delle migliori reti da analizzare partendo dai risultati parziali ottenuti da ogni singolo processore (file `merge_results`).
  - Funzioni generali per l'inizializzazione dell'ambiente MPI (file `mpi_functions`).
- La cartella `variables` che contiene il file `static_variables.cpp` in cui sono contenute alcune importanti variabili di tipo *static*.

Nella cartella `libraries` vi è:

- La cartella `constants` che contiene il file `constants.cpp` con le varie costanti di progetto usate.
- La cartella `exploration` che contiene funzioni relative al modo in cui si effettua l'esplorazione dello spazio delle soluzioni. In particolare si ha:
  - `exploration.cpp`: Contiene tutte le funzioni generali sull'esplorazione dello spazio delle soluzioni.
  - `objective_function.cpp`: Contiene tutte le funzioni relative al calcolo della funzione obiettivo (vincoli di fattibilità, di budget e valore della funzione obiettivo).
- La cartella `objects` che contiene gli oggetti che sono stati implementati per gestire le diverse entità del progetto:
  - `line.cpp`: Rappresenta una linea di autobus, con l'id della linea e le fattibilità delle diverse architetture.
  - `electrical_arch.cpp`: Oggetto che racchiude tutti i costi relativi ad un'architettura elettrica.
  - `traditional_arch.cpp`: Oggetto che racchiude tutti i costi relativi ad un'architettura tradizionale (diesel).
  - `network.cpp`: Oggetto che permette di considerare una rete, ossia un insieme di linee di autobus, ognuna elettrificata con una specifica architettura.
- La cartella `print` contiene tutte le funzioni relative a stampe di debug o di output:
  - `print_functions.cpp`: Contiene le funzioni di stampa eseguite nello stdout, per poter principalmente vedere l'andamento del programma.
  - `write_functions.cpp`: Contiene funzioni di scrittura su file di informazioni specifiche dell'esecuzione del programma.
- La cartella `util` contiene diversi tipi di utilities per il programma:
  - `computation_size.cpp`: Contiene funzioni che permettono di monitorare quale è la grandezza effettiva dello spazio di soluzioni assegnato e calcolato da un processore.
  - `file_management.cpp`: Contiene funzioni per la gestione dei file, come l'apertura controllata di un file o la creazione di una cartella per la memorizzazione dei file.
  - `initialization.cpp`: Contiene funzioni per l'inizializzazione delle diverse strutture dati necessarie per la corretta esecuzione del programma.
  - `list_network.cpp`: Contiene funzioni per la gestione di liste di oggetti Network.
  - `parse_arguments.cpp`: Contiene le funzioni che permettono di parsare correttamente gli argomenti passati tramite riga di comando o tramite script di tipo .sh.
  - `split.cpp`: Contiene le funzioni che permettono di fare lo split delle stringhe. Utile per la lettura di file csv.
  - `timestamp.cpp`: Contiene funzioni che permettono di gestire il tempo di esecuzione del programma.

5.2.2 Manuale utente

In questa sezione sono riportate le informazioni necessarie per poter sfruttare al meglio il programma realizzato.

Il programma può essere eseguito in locale o direttamente sull'infrastruttura Cresco.

Una volta che si hanno i file contenuti nella cartella `Onebus`, per effettuare la compilazione del programma si deve accedere alla cartella `static_program`, in cui è presente il main del progetto. Da questa posizione si può procedere alla *compilazione del programma*, ad esempio nel seguente modo:

```
mpic++ -std=c++11 -Wall -O3 -g onebus_static_main.cpp -o onebus_static_main.out
```

In questo modo avremo ottenuto il file eseguibile `onebus_static_main.out`.

Per poter eseguire su Cresco è bene crearsi uno script (.sh) per l'esecuzione del programma. Un esempio è riportato di seguito:

```
1.  #!/bin/sh
2.  exe=/path_of_executable/onebus_static_main.out  # path of your MPI program
3.  HOSTFILE=$LSB_DJOB_HOSTFILE    # name of hostfile for mpirun
4.  N_procs=`cat $LSB_DJOB_HOSTFILE | wc -l`  # give to mpirun same number of slots
5.  mpirun --mca plm_rsh_agent "blaunch.sh" -n $N_procs --hostfile $HOSTFILE $exe \
6.  -b 30 -f 3 -t 0.8 -d 5 -n 45 -i /inputFolder/linee_output_best.csv \
7.  -o outputFolder/ -r /inputFolder/percorsi.csv
```

All'interno dello script (riga 6 e 7) vengono specificati i diversi parametri con i quali chiamare il programma, come il budget, la funzione di bilanciamento e altri.

È bene conoscere i diversi parametri che si possono specificare per poter sfruttare al meglio il programma e poter eseguire diversi tipologie di test:

- ***-b*** è l'opzione che permette di specificare il *budget* dedicato agli investimenti.
- ***-c*** è l'opzione che permette di specificare il numero di *core* il cui carico assegnato dall'algoritmo *firstLessLastMore* deve essere ulteriormente diviso tramite l'algoritmo di bilanciamento in uso. Tale opzione deve essere specificata solo se l'algoritmo è di bilanciamento in profondità.
- **-d** permette di indicare il *degree* con il quale effettuare il bilanciamento in profondità.
- **-f** indica la funzione di bilanciamento scelta per il run:
  - 0 indica il bilanciamento equo di linee.
  - 1 indica il bilanciamento firsLessLastMore.
  - 2 indica il bilanciamento fisso in profondità.
  - 3 indica il bilanciamento variabile in profondità.
- **-h** per avere in stampa tutte le indicazioni sull'uso dei parametri.
- **-i** per indicare il file di input principale. Si tratta del file contenente le informazioni delle linee che sono state estrapolate tramite il programma BEST (linee_output_best.csv).
- **-n** per specificare il numero di reti migliori che si devono restituire in output.
- **-o** per specificare la cartella in cui si andranno a memorizzare tutti i file di output generati dal programma.
- **-r** per specificare il file dei percorsi che si deve considerare (percorsi.csv).
- **-t** per indicare la tolleranza usata per specificare il vincolo di budget.

Una volta che si è scritto lo script con i parametri corretti si può eseguire su di una delle code disponibili su Cresco con il comando bsub. Un esempio è il seguente comando:

```
bsub -n 16 -o output.out -e error.err -q cresco4_h6 ./script.sh
```

Se l'esecuzione del programma avviene con successo, nella cartella di output ci si ritroverà con:

- `best_networks_id.txt` un file di output per ogni processore id, che indica le migliori sottoreti che sono state analizzate da quel processore.

- `Output_id.txt` in cui sono riportate le statistiche sul numero di sottoreti assegnate e analizzate da quel processore.
- Un file csv con il nome che segue un formato particolare con l'elenco delle migliori sottoreti da elettrificare secondo la funzione obiettivo. Per comprendere meglio ne riportiamo un esempio di seguito.

Esempio: *l30_c10_b90.000000_t0.800000.csv* è il file dove viene memorizzato l'output del test fatto su di una rete con 30 linee, 10 capolinea, budget 90 e tolleranza pari a 0.8. Quando si apre il file si ha la seguente schermata:

obj_function_value	number_of_lines	line_id	arch_id	line_id	arch_id	line_id	arch_id	line_id	arch_id	line_id	arch_id	line_id	arch_id	line_id	arch_id
-6.859.656.867	5	4	A	8	A	17	A	19	B	29	A				
-6.968.213.855	5	4	A	8	A	17	A	28	B	29	A				
-7.000.000.000	6	4	A	8	A	14	A	17	A	28	A	29	A		
-7.200.000.000	6	4	A	8	A	17	A	20	A	28	A	29	A		
-7.200.000.000	6	4	A	8	A	17	A	19	A	28	A	29	A		
-7.300.000.000	6	4	A	8	A	11	A	14	A	17	A	29	A		
-7.400.000.000	6	4	A	8	A	12	A	17	A	28	A	29	A		
-7.400.000.000	5	4	A	8	A	12	A	14	A	17	A				
-7.400.000.000	7	4	A	8	A	9	A	11	A	17	A	28	A	29	A
-7.459.656.867	6	4	A	11	A	17	A	19	B	28	A	29	A		
-7.500.000.000	6	4	A	8	A	17	A	28	A	29	A	30	A		
-7.500.000.000	6	4	A	8	A	17	A	23	A	28	A	29	A		
-7.500.000.000	6	4	A	8	A	11	A	17	A	20	A	29	A		
-7.500.000.000	6	4	A	8	A	11	A	17	A	19	A	29	A		
-7.500.000.000	6	4	A	8	A	9	A	12	A	17	A	29	A		

Figura 5.1: In questa figura viene riportato l'esempio di un file di output in cui vengono specificate le migliori sottoreti che si possono elettrificare.

In tale file vengono riportate le 15 migliori soluzioni possibili e sono specificati:

- Valore della funzione obiettivo.
- Grandezza della sottorete da elettrificare.
- Elenco di id linea e architettura con cui specificare quella linea.

6 Test preliminari e valutazioni sperimentali

Dalle trattazioni effettuate nei precedenti capitoli, si può comprendere come in fase di test degli algoritmi entrino in gioco diversi fattori e molteplici variabili che possono far avere successo ad un test, o farlo fallire. Infatti, il successo di un run dipende da molteplici fattori:

- L'istanza del problema, ossia il numero di linee degli autobus della rete considerata, il budget che si ha a disposizione, la tolleranza che si è disposti ad accettare sul costo degli investimenti. Ma soprattutto la distribuzione dei valori dei costi di investimento, in quanto incidono sul vincolo budget.
- La funzione di bilanciamento e i parametri scelti per quell'algoritmo.
- L'architettura sottostante sulla quale è stato effettuato il run (CRESCO 3 o 4) e il tempo a disposizione per il run.

Nelle successive sezioni verranno mostrati test che permettono di capire come tali parametri incidano sull'esecuzione del programma. Per ogni test fatto sono riportati i seguenti elementi:

- Una breve descrizione del test.
- i dati in formato tabulare.
- i dati in formato grafico.

Legenda:

Tabella 6.1: Legenda per i parametri usati nei test.

LEGENDA	
Simbolo	Significato
lines	Numero di linee degli autobus della rete analizzata
budget	Il budget massimo (in M€) per gli investimenti iniziali
tol	Tolleranza accettata per gli investimenti iniziali
alg	Algoritmo di bilanciamento
f0	Funzione di bilanciamento di linee equo.
f1	Funzione di bilanciamento firstLessLastMore
f2	Funzione di bilanciamento fisso in profondità
f3	Funzione di bilanciamento variabile in profondità
degree	Il grado di suddivisione del carico scelto per gli algoritmi f2 e f3
cores	Numero di core usati per l'esperimento
corBal	I core per cui il carico è stato bilanciato in base al degree
env	Environment sul quale sono stati eseguiti i test
runLim	Il tempo massimo per il run del test

In questa sezione si sono effettuati dei test per comprendere al meglio le potenzialità del codice. Le istanze usate, sono state create ad hoc, cercando di creare delle reti con linee diversificate, secondo 5 fasce di costi (bassissimo, basso, medio, alto e altissimo).

I test riportati in questa sezione sono stati effettuati su di una versione del codice che simula anche l'analisi di un'architettura C di ricarica, pur non tenendo conto di un eventuale fattore di riduzione dei costi dovuto all'effetto rete. Si tratta di una versione del codice leggermente diversa da quella finale, ma i cui test sono stati fondamentali per capire le potenzialità di questo programma.

Invece, i test che si riporteranno nel capitolo successivo, sono stati eseguiti sulla versione finale del codice che tiene conto solo delle architetture A (deposito) e B (capolinea) di ricarica, in cui è stato implementato il fattore di riduzione dei costi dovuto all'effetto di condivisione dei nodi capolinea.

6.1 Test al variare del tipo di bilanciamento

Il primo test significativo che si intende analizzare è quello che si è eseguito al variare della funzione di bilanciamento. Ovvero si sono mantenuti fissi tutti i parametri, cambiando solamente l'algoritmo di load balancing scelto.

6.1.1 Utilizzo di 16 cores

Questo test è stato effettuato su CRESCO 4 con run limit pari a 6 ore.

Tabella 6.2: Test al variare dell'algoritmo di bilanciamento con 16 cores.

Test al variare del tipo di bilanciamento		
Parametri: lines = 50, budget = 50, cores = 16, corBal = 16, env = CRESCO4		
alg	degree	timeExec (s)
f0	-	12225,9
f1	-	10542,8
f2	3	2512,34
f3	4	2040,22

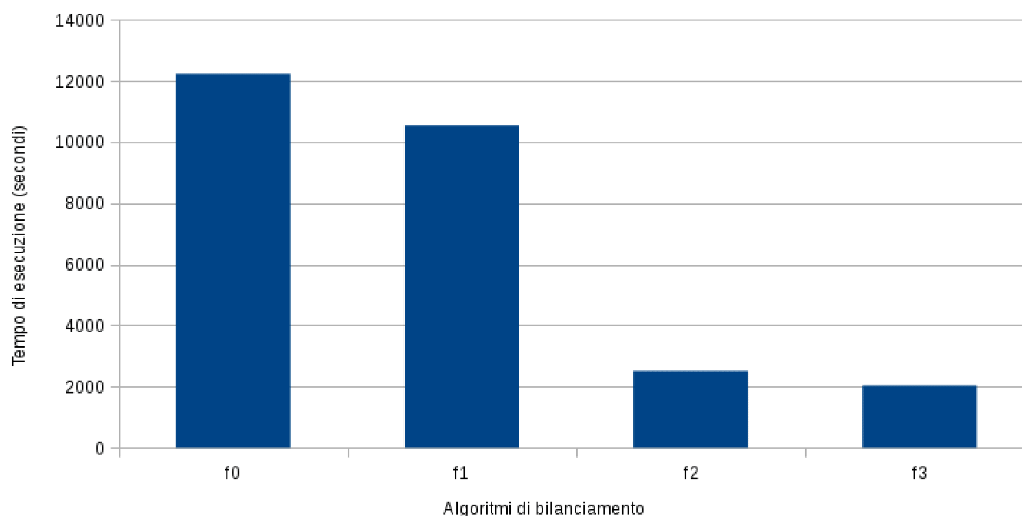

Figura 6.1: Test al variare dell'algoritmo di bilanciamento con 16 cores.

Come si può notare dalla tabella 5.2 e dal grafico in figura 5.1, i tempi di esecuzione degli algoritmi con bilanciamento (sia fisso che variabile) in profondità sono molto minori rispetto ai due algoritmi di *bilanciamento equo* e *firstLessLastMore*. Per comprendere le reali motivazioni di questi miglioramenti, nelle successive sezioni verranno mostrati il numero di sottoreti assegnate, e la distribuzione della grandezza delle reti esplorate da un processore.

6.1.2 Sottoreti assegnate e analizzate da ogni processore

In questa sottosezione si vuole mostrare la distribuzione del numero di sottoreti che gli algoritmi di bilanciamento assegnano ad ogni singolo core, e poi il numero delle effettive sottoreti analizzate da quel core.

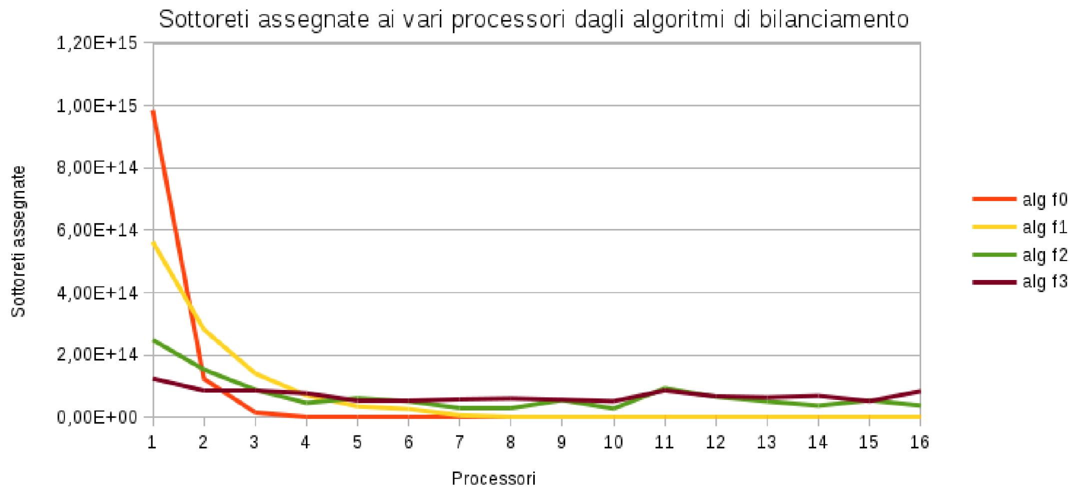

Figura 6.2: Numero di reti assegnate ad ogni processore. Tale numero non tiene conto dell'eventuale taglio dello spazio di soluzioni che si effettua in fase di esplorazione della rete. Si tratta dunque di un limite massimo di reti analizzabili per singolo processore.

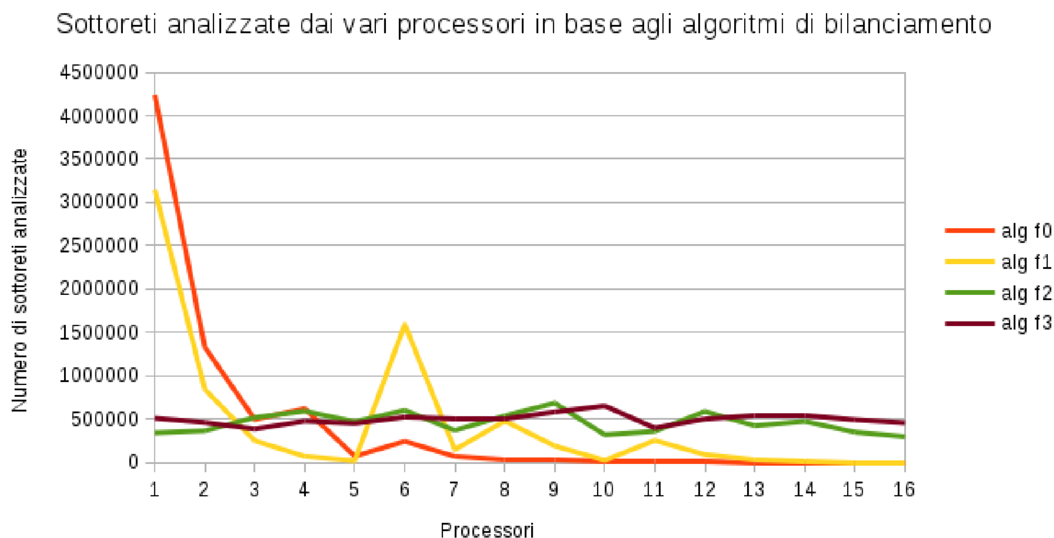

Figura 6.3: Numero effettivo di reti analizzate da ogni processore in fase di esplorazione della rete. Tale numero tiene conto dei tagli dello spazio delle soluzioni.

Come è evidente dai grafici riportati, nel caso dei primi due algoritmi, il carico di sottoreti assegnate e analizzate non è uniforme. Infatti, si nota come il carico che i primi processori devono gestire sia esponenzialmente più grande del carico gestito dagli ultimi processori.

Confrontando l'algoritmo equal mode e firstLessLastMore, si nota una lieve differenza nel carico dei primi processori, a vantaggio di firstLessLastMore. Tuttavia, il carico risulta troppo sbilanciato in entrambi i casi. Invece, analizzando gli algoritmi di bilanciamento in profondità, si può notare come il numero di sottoreti assegnate e analizzate, a meno di qualche fluttuazione, risulta essere molto bilanciato.

Nonostante ciò, l'analisi dei successivi test, mostrerà come ci sono risultati apparentemente inaspettati. Infatti, seppur i processori esplorano un numero di sottoreti pressoché simile, ad incidere sui tempi di esecuzione vi è anche il fattore della grandezza di tali sottoreti analizzate. Per comprendere ciò riportiamo nella tabella 6.3 i valori dei tempi di esecuzione registrati dall'algoritmo di bilanciamento variabile in profondità.

Tabella 6.3: Test al variare dell'algoritmo di bilanciamento con 16 cores.

Test algoritmo di bilanciamento variabile in profondità			
lines = 50, budget = 50, cores = 16, corBal = 16, degree = 4			
core	reti assegnate	reti analizzate	timeExec (s)
1	124366737741009	508897	1190,34
2	85810977993146	459149	939,095
3	85243776659766	383774	447,377
4	77296777406063	475535	780,997
5	51947232284209	449360	729,246
6	54548780981145	522154	1003,79
7	57607776857695	510194	1051,1
8	60272618610377	493654	1352,18
9	56241963800829	579510	2040,15
10	51282093769903	649955	1379,27
11	86103888599067	396403	487,213
12	67052453026414	499128	1653,73
13	63733860059413	530564	1692,52
14	68614826942837	540688	980,938
15	51776376436282	491383	682,049
16	83999766674650	455619	835,308

La tabella 6.3 mostra due risultati che potrebbero sorprendere:

1. Pur essendo state assegnate ad un processore più sottoreti rispetto ad un altro, può accadere che tale processore analizzi meno sottoreti dell'altro processore. È il caso che si può vedere in questa tabella con i processori 1 e 9. Il primo core ha assegnato più di 124 mila miliardi di sottoreti, ma in fase di esplorazione (grazie al branch and bound) ne analizza poco più di 508 mila. Invece, il core numero 9, ne ha assegnate 56 mila miliardi, ma ne analizza più di 579 mila. Questo accade perché l'algoritmo di branch and bound riesce a tagliare un numero di soluzioni diverso in base alle porzioni di rete che sta analizzando.
2. Pur avendo analizzato un numero minore di sottoreti, il tempo di esecuzione di un processore può essere peggiore di quello di un altro processore che analizza più sottoreti. È il caso dei processori 8 e 14. Il processore 8 analizza più di 493 mila sottoreti, impiegando 1352,18 secondi per analizzarle. Invece, il processore 14 ne analizza 540 mila, ma impiega meno tempo (980,938 secondi). Questo risultato è legato al fatto che il processore 8, esplora meno reti rispetto al processore 14, ma analizza sottoreti più grandi, e questo incide enormemente sul tempo di esecuzione degli algoritmi. Per comprendere meglio questo tipo di evento, riportiamo la distribuzione delle grandezze delle sottoreti analizzate da questi due processori in figura 6.4.

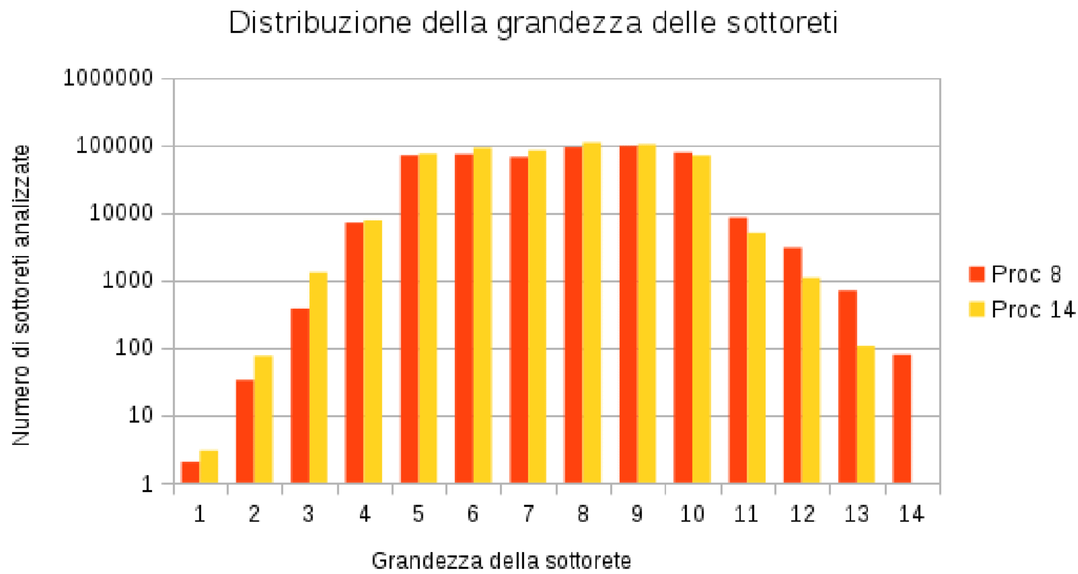

Figura 6.4: Distribuzione della grandezza delle sottoreti analizzate dai processori 8 e 14.

Il grafico 6.4 mostra come il processore 8 analizza più reti di grandi dimensioni, rispetto al processore 14. E questo è il motivo per cui il tempo di esecuzione è maggiore, nonostante complessivamente analizza meno sottoreti.

6.1.3 Utilizzo di 32 cores

Questo test è stato effettuato su CRESCO 4 con run limit pari a 6 ore.

Tabella 6.4: Test al variare dell'algoritmo di bilanciamento con 32 cores.

Test al variare del tipo di bilanciamento		
lines = 50, budget = 50, cores = 32, corBal = 32, env = CRESCO4		
alg	degree	timeExec (s)
f0	-	10444,1
f1	-	10484,4
f2	5	1305,34
f3	5	1182,42

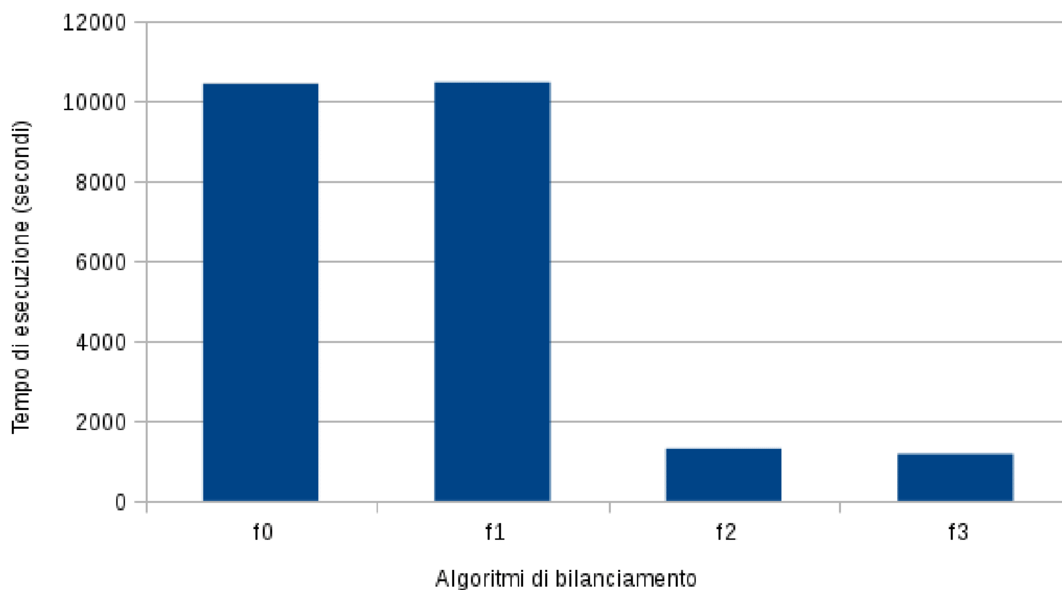

Figura 6.5: Test al variare dell'algoritmo di bilanciamento con 32 cores.

Questo test mostra degli interessanti risultati:

- Il primo risultato riguarda gli algoritmi f0 e f1, il cui tempo di esecuzione è molto simile, al contrario di quanto era accaduto nel caso con 16 cores. Il motivo per cui l'algoritmo *firstLessLastMore* non riesce a migliorare il tempo di esecuzione è dovuto al fatto che in questo caso con 32 cores e 50 linee si comporta proprio come l'algoritmo *equal mode*. Infatti, i primi processori ricevono comunque una sola linea di cui calcolare le possibili combinazioni di sottoreti.
- Il secondo risultato è che il divario tra i tempi di esecuzione dei due algoritmi di bilanciamento in profondità è minore rispetto al caso con 16 cores. Il motivo è legato al fatto che nel caso di 16 cores si sono testati i due algoritmi con un valore di degree diverso (degree pari 3 per f2 e degree pari a 4 per f3). Invece, in questo test si è usato il valore di degree = 5 per entrambi gli algoritmi. 3.
- Il terzo risultato è che a parità di degree l'algoritmo di bilanciamento variabile in profondità, mostra dei risultati migliori rispetto al bilanciamento fisso. Questo è legato al fatto che si evita di suddividere inutilmente il carico di una linea, quando il workload non è così pesante. Ma si attua un'ampia suddivisione del carico per le linee effettivamente più onerose da calcolare.

6.2 Test al variare del numero di linee

In questo test si è voluto provare a far crescere il numero di linee di autobus che compongono la rete analizzata. Si è effettuato questo test per due valori di budget: 45M€ e 20M€.

6.2.1 Utilizzo di budget medio-grande e 32 cores

Questo test è stato effettuato con un budget di 45M€ sull'architettura di CRESCO 3 con 32 cores, in quanto solo su CRESCO 3 si è avuto accesso a code con runLimit anche di 6 giorni.

Tabella 6.5: Test al variare del numero di linee con budget medio grande.

Test al variare del numero di linee	
budget = 45, degree = 4, alg = f3, cores = 32, env = CRESCO3	
lines	timeExec (s)
15	0,19
30	1,45
45	215,62
60	12327,8
75	exceed

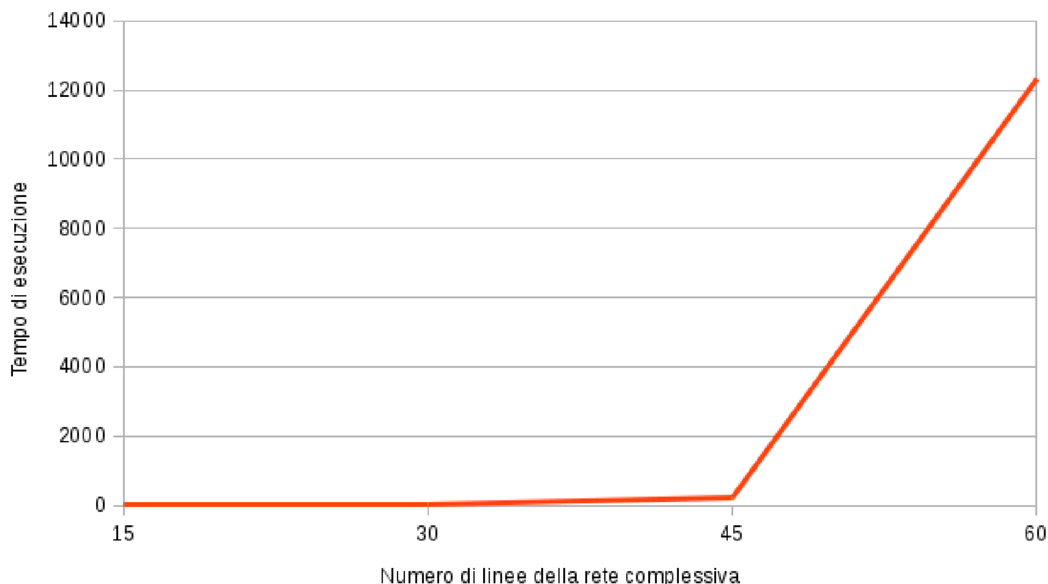

Figura 6.6: Andamento del tempo di esecuzione al variare del numero di linee presenti nell'istanza di ingresso con budget pari a 45M€.

Il test mostra come non sia stato possibile analizzare una rete di 75 linee. Il motivo di questo risultato è legato al fatto che crescendo il numero di linee, aumenta la complessità del calcolo. Infatti, non solo aumentano il numero di sottoreti, ma aumenta anche la probabilità di dover analizzare sottoreti composte da più linee, in quanto è verosimile che essendoci più linee complessivamente, alcune di queste siano a basso costo di investimento. In questo modo, la combinazione di più linee a basso costo di investimento rientra nel vincolo di budget, e ciò porta a dover analizzare reti di dimensione maggiore, con la conseguente crescita esponenziale dei tempi di esecuzione.

6.2.2 Utilizzo di budget piccolo e 256 cores

Questo test è stato effettuato con un budget di 20M€ sull'architettura di CRESCO 4 con 256 processori, con un runLimit di 6 ore.

Tabella 6.6: Test al variare del numero di linee con budget piccolo.

Test al variare del numero di linee	
budget = 20, degree = 5, alg = f3, cores = 256, env = CRESCO4	
lines	timeExec (s)
70	53,04
80	168,348
90	462,772
100	1201,66
110	3015,87
120	8873,17
130	15029,9
140	exceed

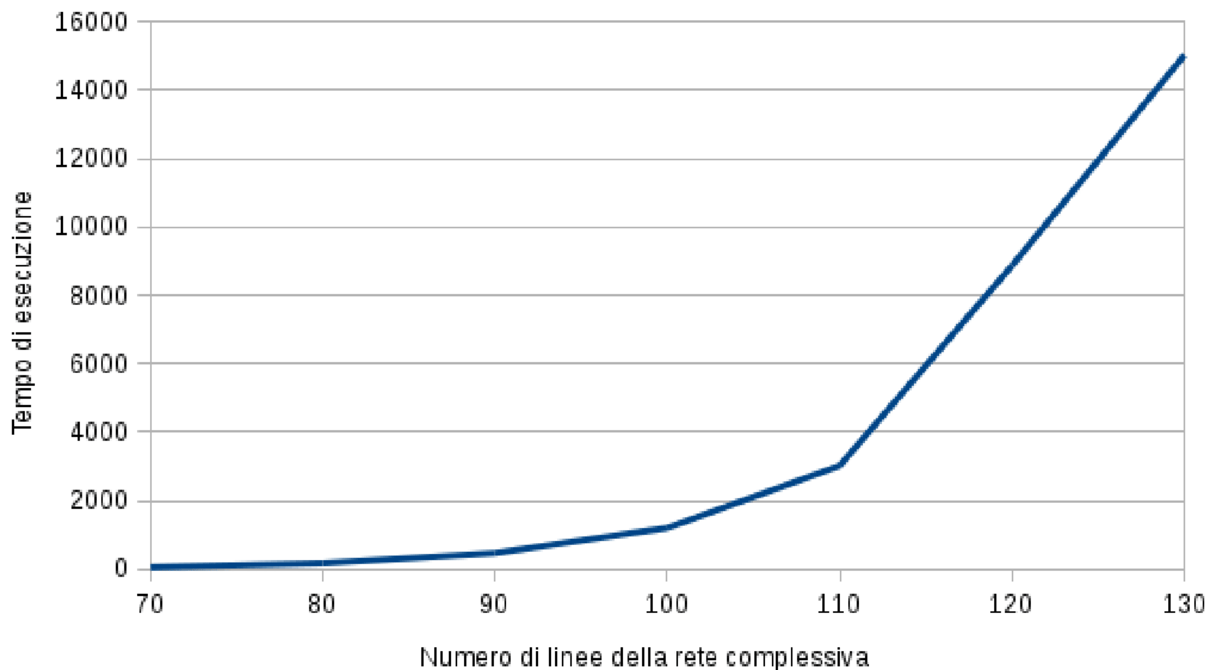

Figura 6.7: Andamento del tempo di esecuzione al variare del numero di linee presenti nell'istanza di ingresso con budget pari a 20M€.

Questo test conferma le conclusioni a cui si era giunti nel test precedente. Diminuendo il valore del budget si è riusciti ad analizzare reti grandi fino a 130 linee.

Dunque, si comprende come il valore di budget sia fondamentale per determinare la complessità dell'esecuzione di questo programma. Più alto è il budget, e più sottoreti rientreranno tra quelle da esplorare, e più aumenta il tempo di esecuzione.

7 Test per il confronto con l'euristica

Tutti i test precedenti ci hanno permesso di conoscere i limiti dell'algoritmo implementato. Per poter confrontare i risultati esatti con quelli euristici si è scelto di generare un set di istanze ad hoc, oltre che di analizzare il caso reale della rete di Firenze.

7.1 Test su istanze ad hoc

Per poter confrontare i risultati esatti con quelli prodotti dall'algoritmo euristico, in primo luogo si è usato un set di istanze create ad hoc: 30 linee al variare del numero di capolinea presenti e al variare dell'istanza i cui dati vengono generati in modo random.

Tali istanze hanno le seguenti proprietà:

- Ogni istanza presenta 30 linee e per ogni singola linea ci sono due percorsi.
- Le voci di costo che caratterizzano le linee sono stati generati in modo randomico con valori interi che variano da 1 a 10.
- La frequenza per ogni percorso è sempre fissa e pari a 10.

Il numero di capolinea viene analizzato solo in questi test finali, perché la realizzazione del calcolo del fattore di riduzione di costo dovuto all'effetto rete, è stato implementato nella versione finale del progetto. Si tratta di un parametro molto importante, in quanto avere un alto numero di capolinea potrebbe rendere più lento il calcolo della funzione obiettivo.

Si sono volute analizzare diverse reti a parità di caratteristiche dell'istanza, per poter valutare la bontà dell'euristica. Tale analisi non è oggetto di questo report, ma per completezza si vuole riportare i tempi di esecuzione degli algoritmi, così da far comprendere il tempo impiegato per trovare le soluzioni che sono state usate successivamente per il confronto.

7.1.1 Istanza con 30 Linee e budget 30

Questi test sono stati effettuati su CRESCO 4 con run limit pari a 6 ore, e utilizzando 16 cores.

Tabella 7.1: Test al variare delle voci di costo e del numero di capolinea, con budget 30.

Test al variare delle voci di costo e del numero di capolinea					
lines = 30, budget = 30, cores = 16, alg = f3, degree = 5, env = CRESCO4, tol = 0.8					
6 Capolinea		10 capolinea		20 capolinea	
TEST	TimeExec (s)	TEST	TimeExec (s)	TEST	TimeExec (s)
1	0.84006	6	0.761931	11	0.851887
2	0.697199	7	0.771638	12	0.795848
3	0.658764	8	0.715212	13	0.805053
4	0.735372	9	0.7736	14	0.928393
5	0.694483	10	0.933163	15	0.881439

7.1.2 Istanza con 30 Linee budget 60

Questi test sono stati effettuati su CRESCO 4 con run limit pari a 6 ore, e utilizzando 16 cores.

Tabella 7.2: Test al variare delle voci di costo e del numero di capolinea, con budget 60.

Test al variare delle voci di costo e del numero di capolinea					
lines = 30, budget = 60, cores = 16, alg = f3, degree = 5, env = CRESCO4, tol = 0.8					
6 Capolinea		10 capolinea		20 capolinea	
TEST	TimeExec (s)	TEST	TimeExec (s)	TEST	TimeExec (s)
16	106.566	21	101.346	26	2275.02
17	24.8583	22	103.056	27	125.039
18	111.952	23	130.465	28	666.705
19	115.753	24	139.578	29	325.376
20	93.2337	25	69.4623	30	109.297

7.1.3 Istanza con 30 Linee e budget 90

Questi test hanno impiegato un tempo di esecuzione superiore ai precedenti, e per poter ottenere dei risultati in tempo si sono eseguiti sia su CRESCO 3 che su CRESCO 4, con run limit massimo pari a 1 giorno e utilizzando almeno 128 cores.

Tabella 7.3: Test al variare delle voci di costo e del numero di capolinea, con budget 90.

Test al variare delle voci di costo e del numero di capolinea							
lines = 30, budget = 90, alg = f3, degree = 5, tol = 0.8							
6 Capolinea				10 capolinea			
TEST	env	cores	TimeExec (s)	TEST	env	cores	TimeExec (s)
31	CRESCO4	128	9318.71	36	CRESCO3	288	36266.5
32	CRESCO4	256	38872.7	37	CRESCO3	288	40969
33	CRESCO4	128	4320.74	38	CRESCO3	288	24859.4
34	CRESCO4	128	8392.2	39	CRESCO4	256	46772.7
35	CRESCO4	128	5245.26	40	CRESCO3	288	58088
20 Capolinea							
TEST	env	cores	TimeExec (s)				
41	CRESCO4	128	6378.4				
42	CRESCO4	128	9493.77				
43	CRESCO4	128	4120.37				
44	CRESCO4	256	29242.1				
45	CRESCO4	128	11424.3				

7.1.4 Analisi dei risultati di test fatti su istanze ad hoc

Questi 45 test sono stati effettuati su istanze di reti diverse. Ciò che dunque si può evincere è che anche a parità di linee e di numero di capolinea, tuttavia la distribuzione delle voci di costo influisce enormemente sulle prestazioni. Ci sono stati test come il 32 e 44 che dopo 6 ore di esecuzione su CRESCO 4 con 128 cores dedicati, non hanno terminato con successo l’esecuzione. Tali test sono stati ripetuti sempre su CRESCO 4 ma con 256 cores e 24 ore di tempo massimo per il run. Questo ha consentito di ottenere il risultato cercato.

Questo esempio mostra chiaramente come nonostante il numero di linee e di capolinea fosse lo stesso, si è impiegato molto più tempo per l'esecuzione del programma. Il motivo anche in questo caso risiede nel fatto che la distribuzione dei costi in questi casi è stata tale da effettuare meno tagli dello spazio delle soluzioni, con la conseguenza di aver dovuto analizzare delle reti di grandezza maggiore.

7.2 Test sul caso reale della rete di Firenze

I test più interessanti sono stati effettuati sulla rete del trasporto pubblico di Firenze. Si tratta di una rete costituita da 85 linee e 101 capolinea. Si è cercato di analizzare i risultati di output al variare del budget. I test riportati sono stati eseguiti con budget a partire da 2 milioni, fino ad un budget di 8 milioni di euro.

Tabella 7.4: Test sulla rete di Firenze al variare del budget.

Test sulla rete di Firenze al variare del budget		
Lines = 85, capolinea = 101, alg = f3, degree = 5,		
Budget (€)	tol	timeExec (s)
2000000	0.9	1.98719
3000000	0.9	2.77997
4000000	0.1	26.2314
4000000	0.9	26.5262
5000000	0.1	301.788
5000000	0.9	309.652
6000000	0.1	2878.09
6000000	0.9	2843.34
7000000	0.9	21216.6

obj_function_value	number_of_lines	line_id	arch_id	line_id	arch_id	line_id	arch_id
989989,1008	1	69	B				
949226,2008	2	18	A	69	B		
938656,3088	2	16	A	69	B		
935939,1355	2	62	A	69	B		
919776,1778	2	32	A	69	B		
616235,0101	1	73	B				
557424,5628	1	7	B				
516661,6628	2	7	B	18	A		
506091,7708	2	7	B	16	A		
503374,5975	2	7	B	62	A		
499653,2168	2	7	B	30	A		
487211,6398	2	7	B	32	A		
469857,8288	2	7	B	40	A		
465328,8708	3	7	B	16	A	18	A
464428,4518	2	7	B	48	A		

Figura 7.1: Risultato test su Firenze con budget 4 milioni di euro e tolleranza 0.1.

obj_function_value	number_of_lines	line_id	arch_id	line_id	arch_id	line_id	arch_id
989989,1008	1	69	B				
949226,2008	2	18	A	69	B		
938656,3088	2	16	A	69	B		
935939,1355	2	62	A	69	B		
919776,1778	2	32	A	69	B		
616235,0101	1	73	B				
516661,6628	2	7	B	18	A		
506091,7708	2	7	B	16	A		
503374,5975	2	7	B	62	A		
499653,2168	2	7	B	30	A		
487211,6398	2	7	B	32	A		
469857,8288	2	7	B	40	A		
465328,8708	3	7	B	16	A	18	A
464428,4518	2	7	B	48	A		
462611,6975	3	7	B	18	A	62	A

Figura 7.2: Risultato test su Firenze con budget 4 milioni di euro e tolleranza 0.9.

obj_function_value	number_of_lines	line_id	arch_id	line_id	arch_id	line_id	arch_id
1029961,274	2	10	A	69	B		
989989,1008	1	69	B				
949338,9138	2	69	B	84	A		
949226,2008	2	18	A	69	B		
938656,3088	2	16	A	69	B		
935939,1355	2	62	A	69	B		
932217,7548	2	30	A	69	B		
928200,4438	2	69	B	74	A		
926428,7048	2	19	A	69	B		
922691,0838	2	42	A	69	B		
922536,3998	2	39	A	69	B		
919776,1778	2	32	A	69	B		
912950,9538	2	8	A	69	B		
902422,3668	2	40	A	69	B		
897893,4088	3	16	A	18	A	69	B

Figura 7.3: Risultato test su Firenze con budget 5 milioni di euro e tolleranza 0.1.

obj_function_value	number_of_lines	line_id	arch_id	line_id	arch_id	line_id	arch_id
1029961,274	2	10	A	69	B		
949338,9138	2	69	B	84	A		
928200,4438	2	69	B	74	A		
922691,0838	2	42	A	69	B		
912950,9538	2	8	A	69	B		
891454,8548	3	18	A	30	A	69	B
885665,8048	3	18	A	19	A	69	B
881773,4998	3	18	A	39	A	69	B
880884,9628	3	16	A	30	A	69	B
878167,7895	3	30	A	62	A	69	B
875095,9128	3	16	A	19	A	69	B
872378,7395	3	19	A	62	A	69	B
871203,6078	3	16	A	39	A	69	B
868486,4345	3	39	A	62	A	69	B
865117,1988	2	22	A	69	B		

Figura 7.4: Risultato test su Firenze con budget 5 milioni di euro e tolleranza 0.9.

obj_function_value	number_of_lines	line_id	arch_id	line_id	arch_id	line_id	arch_id
1266391,262	2	69	B	70	B		
1225628,362	3	18	A	69	B	70	B
1215058,47	3	16	A	69	B	70	B
1212341,297	3	62	A	69	B	70	B
1196178,339	3	32	A	69	B	70	B
1130534,963	2	14	B	69	B		
1098094,074	2	4	B	69	B		
1089772,063	3	14	B	18	A	69	B
1079202,171	3	14	B	16	A	69	B
1076484,998	3	14	B	62	A	69	B
1075941,695	2	61	A	69	B		
1066583,005	2	69	B	77	A		
1063893,162	2	45	B	69	B		
1060322,04	3	14	B	32	A	69	B
1057331,174	3	4	B	18	A	69	B

Figura 7.5: Risultato test su Firenze con budget 6 milioni di euro e tolleranza 0.1.

obj_function_value	number_of_lines	line_id	arch_id	line_id	arch_id	line_id	arch_id
1266391,262	2	69	B	70	B		
1225628,362	3	18	A	69	B	70	B
1215058,47	3	16	A	69	B	70	B
1212341,297	3	62	A	69	B	70	B
1196178,339	3	32	A	69	B	70	B
1098094,074	2	4	B	69	B		
1089772,063	3	14	B	18	A	69	B
1079202,171	3	14	B	16	A	69	B
1076484,998	3	14	B	62	A	69	B
1075941,695	2	61	A	69	B		
1066583,005	2	69	B	77	A		
1063893,162	2	45	B	69	B		
1060322,04	3	14	B	32	A	69	B
1057331,174	3	4	B	18	A	69	B
1051917,775	2	69	B	83	A		

Figura 7.6: Risultato test su Firenze con budget 6 milioni di euro e tolleranza 0.9.

obj_function_value	number_of_lines	line_id	arch_id	line_id	arch_id	line_id	arch_id	line_id	arch_id
1547413,66364208	2	7	B	69	B				
1336217,80579938	3	18	A	52	B	69	B		
1325647,91379938	3	16	A	52	B	69	B		
1322930,74049938	3	52	B	62	A	69	B		
1319209,35979938	3	30	A	52	B	69	B		
1313420,30979938	3	19	A	52	B	69	B		
1309528,00479938	3	39	A	52	B	69	B		
1306767,78279938	3	32	A	52	B	69	B		
1306363,43491437	3	10	A	69	B	70	B		
1289413,97179938	3	40	A	52	B	69	B		
1284885,01379938	4	16	A	18	A	52	B	69	B
1283984,59479938	3	48	A	52	B	69	B		
1282167,84049938	4	18	A	52	B	62	A	69	B
1281425,77779938	3	52	B	55	A	69	B		
1278399,93679938	3	50	A	52	B	69	B		

Figura 7.7: Risultato test su Firenze con budget 7 milioni di euro e tolleranza 0.9.

Con questi test si è potuto capire quali fossero le migliori 15 sottoreti da poter elettrificare nella rete di Firenze avendo un budget fino a 7 milioni di euro. I test hanno evidenziato che:

- Con un budget di 7 milioni di euro, le migliori sottoreti che si riescono ad elettrificare contengono 2, 3 o 4 linee.

- La tolleranza non è un fattore così determinante. Una tolleranza piccola implica un numero maggiore di possibili soluzioni che rientrano nel vincolo del budget, ma il vero fattore determinante non è il budget minimo (dipende dalla tolleranza), bensì il budget massimo. E qualunque sia il valore della tolleranza, il budget massimo rimane invariato.
- In tutti i casi si ottiene un valore della funzione obiettivo positiva, che quindi individua come sia possibile iniziare il processo di passaggio ad alimentazione elettrica, ottenendo comunque dei guadagni economici dal passaggio tra le due tecnologie. Ad esempio dall'elettrificazione di due sole linee si avrebbe un guadagno di progetto pari a 1˙547˙413,66€.
- L'aumento del tempo di esecuzione è legato ai fattori che sono stati analizzati nei precedenti test:
  - Con l'aumento del budget si ottiene un aumento del numero di sottoreti che rientrano nel vincolo di budget, ciò comporta un aumento del tempo impiegato per l'analisi complessiva dello spazio delle soluzioni, perché non si riescono a tagliare molte possibili soluzioni.
  - Inoltre aumenta anche la grandezze delle sottoreti analizzate e questo aumenta di molto il tempo di esecuzione dell'algoritmo di esplorazione.
- Si nota come all'aumentare del budget massimo, aumenta anche il valore della funzione obiettivo, segno che maggiori sono gli investimenti nella rete, e maggiori saranno i guadagni conseguenti all'utilizzo dell'alimentazione elettrica.
- Si sono effettuati anche test al variare della tolleranza di budget (*tol = 0.1* e *tol = 0.9*), per capire come cambiano i risultati pur mantenendo il budget fisso. Si evince che molte volte limitando l'intervallo dei costi di investimento (ossia usando *tol = 0.9*) si possono escludere delle soluzioni che sono maggiormente convenienti di altre. Un esempio è legato al caso con budget 5 milioni: si nota come con *tol = 0.9* viene esclusa la soluzione con la rete formata dalla sola linea 69 elettrificata con architettura B, nonostante tale soluzione sarebbe stata migliore della seconda soluzione trovata.

8 Conclusioni

Gli esperimenti e lo studio teorico riportato in questo report hanno messo in evidenza come non sia stato semplice riuscire a risolvere un problema così vasto e di natura np-hard.

L'attività svolta dall'Università di Tor Vergata ha portato alla creazione di algoritmi in grado di poter essere eseguiti su di architetture parallele come quella di CRESCO e in grado di poter suddividere il carico di lavoro tra diversi processori.

Si è visto come grazie allo studio teorico del bilanciamento del carico sia stato possibile passare da algoritmi molto semplici e poco efficaci, ad altri più complicati ma molto più efficienti. Si tratta di algoritmi in grado di utilizzare tecniche di branch and bound, senza le quali non sarebbe stato possibile esplorare anche reti molto piccole. Invece, grazie a questi ultimi si sono riuscite ad analizzare reti composte anche da 130 linee, e si sono ottenuti risultati interessanti anche sulla rete di Firenze costituita da 85 linee.

Aver realizzato un algoritmo in grado di trovare sempre le migliori soluzioni ha reso possibile il confronto con l'algoritmo euristico implementato dall'Università degli studi Roma tre. Inoltre, si è dimostrato che il passaggio dal sistema di alimentazione tradizionale a quello elettrico, risulta essere vantaggioso nel caso dell'istanza reale di Firenze.

8.1 Sviluppi futuri

Lo studio che si è realizzato ha mostrato come la difficoltà principale del problema sia stata la ricerca di un buon algoritmo di suddivisione del carico. Compreso come sia possibile suddividere il carico di lavoro in piccole porzioni, il problema da risolvere su ogni singola parte rimane sempre lo stesso.

Si è dunque di fronte ad un problema in cui si ha un unico programma con diversi input. Una tecnologia in grado di sfruttare al meglio le risorse di CRESCO e che ci permetterebbe di poter risolvere istanze del problema maggiormente significative, è il job array.

Infatti, con LSF è possibile sottomettere jobs multipli in un unico run o step, usando un "Job array", ossia un insieme di oggetti dello stesso tipo, cono lo stesso nome, identificati da un indice, ma che assumono valori diversi (per i jobs di un array la differenza è nell'input al programma sottomesso). Questo approccio è particolarmente utile se i job dell'array condividono lo stesso eseguibile (ad es. uno script o un'applicazione) e tipo di risorsa, ma hanno differenti file di input oppure l'eseguibile può gestire una variabile associata ad un indice.

Ciò che migliorerebbe rispetto all'approccio che si è implementato, è che LSF schedulerebbe e distribuirebbe i jobs dell'array indipendentemente l'uno dall'altro, in quanto pur condividendo l'ID ogni job è identificato univocamente da un indice.

Per tali motivi l'implementazione di tali algoritmi di esplorazioni usando il job array è il principale sviluppo futuro per migliorare la ricerca effettuata in questo anno.

9 Riferimenti bibliografici

1. *"Introduction to Parallel Computing"*, Lawrence Livermore National Laboratory, https://computing.llnl.gov/tutorials/parallel_comp/
2. *"MPI Documents"*, http://mpi-forum.org/docs/
3. M. Kushwaha, S. Gupta *"Various Schemes of Load Balancing in Distributed Systems- A Review"*, July 2015 (IJSRET), ISSN 2278 - 0882

10 Abbreviazioni ed acronimi

CRESCO: Computational RESearch centre on Complex system.
DICII: Dipartimento di Ingegneria Civile ed Ingegneria Civile.
ENEA: Agenzia nazionale per le nuove tecnologie, l'energia e lo sviluppo economico sostenibile.
HPC: High performance computing.
TPL: Trasporto pubblico locale.
MPI: Message Passing Interface.
Lines: Numero di linee degli autobus di una rete.
Tol: Tolleranza accettata per gli investimenti iniziali.
Alg: algoritmo di bilanciamento.
f0: Funzione di bilanciamento di linee equo.
f1: Funzione di bilanciamento firstLessLastMore.
f2: Funzione di bilanciamento fisso in profondità.
f3: Funzione di bilanciamento variabile in profondità.
corBal: I cores per cui il carico e stato bilanciato in base al degree
env: Environment sul quale sono stati eseguiti i test.
runLim: Il tempo massimo per il run del test.