Università degli Studi di Padova

Ingegneria informatica

Tesina del corso di Ricerca operativa 2

Professore Matteo Fischetti

Ottimizzazione di una wind farm

Autori

Davide Martini

Simone Nigro

Sommario

In questa tesina viene affrontato un problema di ottimizzazione lineare intera. Si mostra lo studio dell'analisi matematica che ci permette di modellizzarlo e successivamente risolverlo. E' scopo di questo elaborato mostrare i vari algoritmi che migliorano l'efficienza dal punto di vista della programmazione. Per nostra scelta, ci concentreremo sull'implementazione di metodi inerenti al software IBM ILOG CPLEX [1] utilizzando il linguaggio C. Infine vengono riportati alcuni algoritmi euristici, trattati durante il corso di Ricerca Operativa 2, che permetteranno un miglioramento in termini di tempo della risoluzione del problema ma ad un possibile peggioramento qualitativo della soluzione ottima.

Indice

Elenco delle figure

Elenco delle tabelle

1

Descrizione del problema e sua modellizzazione

In questo capitolo verrà descritto il problema trattato e la sua modellizzazione matematica, [2]. Infine si presenterà il modo in cui il modello si traduce e si risolve in un problema di programmazione con l'aiuto del software CPLEX.

1.1 Modello matematico

Il problema di ottimizzazione di cui ci occuperemo riguarda una wind farm. Ci vengono fornite delle turbine, già posizionate precedentemente in mare aperto, e una o più substation, ovvero dei punti nel mare dove viene raccolta l'energia elettrica che viene poi trasferita a terra attraverso un cavo. Ipotizziamo ci sia una sola substation ed indicizziamo le turbine da 1 a n. Una prima rappresentazione di questo problema può essere fatta attraverso un grafo completo $n \times n$. Il nostro obiettivo è quello di creare una struttura per la raccolta della corrente, la quale, attraverso un cavo che collega una turbina ad un'altra, conduce la corrente generata alla substation ed infine alla costa. Ogni turbina deve essere collegata in una struttura tale che ci sia un cammino orientato nel quale ciascuna fa arrivare la corrente nella substation. Ci aspettiamo che la soluzione finale disegnata sia rappresentabile da un albero orientato dove la radice di quest'ultimo riceve gli archi.

Tecnicamente questa struttura viene chiamata anti arborescenza. Per poterla definire, è necessario scegliere, per ogni coppia di turbine (i, j), se mettere o meno un collegamento da i verso j.

Seguendo l'approccio descritto in [2], creiamo un modello matematico con delle variabili binarie (1.2) che rappresentano questa scelta e le chiamiamo y_{ij}. Essa vale 1 se costruisco l'arco (i, j) e 0 in caso contrario. Ne abbiamo una per ciascun arco, ossia visto che il grafo è completo ne avremo una per ogni i e j che vanno da 1 a n. In realtà bisognerebbe escludere y_{ii} poiché nessuna turbina è collegata con un cavo a se stessa, ma per evitare di avere un modello più complicato da gestire la includiamo ugualmente e la fissiamo a 0 per ogni i da 1 a n. In questo modo la notazione del modello non si appesantisce e a livello computazionale è trasparente perché una variabile fissata a zero viene subito eliminata da CPLEX in fase di pre-processing.

Definito il set di variabili fondamentali, è ora necessario stabilire, nel modello, che le y devono costruire un'arborescenza e in qualche modo devono supportare tutta l'energia prodotta. Ogni turbina h ha una produzione elettrica, chiamata da noi P_h e assume valore maggiore di zero. Essa rappresenta la corrente generata in un'unità di tempo dalla turbina h. Nei file di input considerati, i valori delle potenze sono tutti uguali e sono scalati in modo da avere $P_h = 1$, se ci riferiamo alla produzione di una turbina, o $P_h = -1$, se indichiamo l'unico punto della rete non generatrice di corrente, ovvero la substation. Altri dati forniti in input sono la coordinata X e la coordinata Y che identificano la posizione geografica di ogni turbina.

Un primo vincolo (1.5), caratteristico del nostro modello, impone un solo arco in uscita da ogni nodo (esclusa la substation). Questo vincolo di grado lo chiamiamo *out degree constraint*: se conto quanti cavi escono da una generica turbina h, usando le variabili y, la somma per j che va da 1 a n degli y_{hj} deve essere uguale a uno se la produzione della turbina è maggiore o uguale a zero, cioè se siamo in una vera turbina, oppure deve essere uguale a zero se il valore di P_h è -1, ovvero se siamo nella substation. Questo vincolo definisce una prima parte del modello nel quale viene semplicemente imposta una proprietà degli archi appartenenti all'arborescenza.

Un secondo vincolo (1.6) indica un numero massimo, c, di bocchettoni in ingresso alla substation. Questa esigenza si traduce graficamente in un limite non superabile sul numero di archi in entrata al nodo h con $P_h = -1$; in teoria tale vincolo riguarderebbe anche le altre turbine ma di solito a livello tecnologico non è molto rilevante. Nel modello scriviamo quindi che la somma delle y_{ih} per i da 1 ad n, ovvero il numero di cavi entranti

nella turbina h, deve essere minore o uguale al valore C, dato in input.
Questo vincolo verrà generato solo una volta nel caso in cui la substation sia unica.
Un altro insieme di variabili chiamate f_{ij} (1.4), continue e maggiori o uguali a zero, rappresentano la quantità di corrente elettrica sull'arco (i, j). La caratteristica di questo flusso è quella di rispettare la legge di Kirchhoff (1.7), se prendiamo una turbina h e calcoliamo il flusso in entrata, sommando tutti i flussi in ingresso, e aggiungiamo la quantità di energia prodotta dalla turbina stessa, otteniamo il flusso in uscita da h. Anche in questo caso f_{ii} non esisterebbe, essendo nullo il flusso che va da una turbina a se stessa, ma per comodità viene posta uguale a zero.
Infine definiamo delle variabili x_{ij}^k (1.3), indicanti l'utilizzo del cavo di tipo k sul collegamento (i, j). I cavi vengono forniti in input, sono di tipi diversi ed ognuno ha un determinato costo al metro e capacità. La variabile x_{ij}^k è di tipo binario e vale 1 se uso il cavo di tipo k sull'arco (i, j), 0 in caso contrario. Non esistendo nessun cavo che collega una turbina a se stessa x_{ii}^k viene posta a zero. Di questo tipo di variabile ne abbiamo una per ogni i,j da 1 ad n e una per ogni tipo di cavo, indicizzati da k compreso tra 1 e K.
A questo punto è necessario creare dei vincoli per mettere in relazione le tre variabili. Una prima dipendenza (1.8) nasce dall'uguaglianza tra y_{ij} e la somma per k da 1 a k degli x_{ij}^k, per ogni i,j da 1 ad n. Questo impone che è possibile scegliere un solo tipo di cavo su un collegamento (i, j) costruito, infatti la variabile y può valere al massimo 1. Grazie a questa relazione esistente avremmo potuto evitare l'utilizzo delle y nel nostro modello, tuttavia in questo modo il problema matematico risulta essere più comprensibile e non ne peggiora la risoluzione.
Per poter scegliere il cavo adatto a supportare la corrente passante da quel collegamento, è necessario introdurre un altro vincolo (1.9): se così non fosse, la soluzione ottima non presenterebbe nessun cavo o sceglierebbe sempre quello più economico. Per le ragioni appena esposte è opportuno porre molta attenzione alla fase di inserimento dei cavi, dove bisogna scegliere quello meno costoso ma con sufficiente capacità in grado di supportare l'energia che lo percorre.
Bisogna quindi garantire che su ogni collegamento la capacità installata sia maggiore o uguale al flusso sull'arco (i, j), cioè a f_{ij}. Tale capacità è calcolabile attraverso la somma, al variare di k da 1 a K, delle x_{ij}^k moltiplicate per la corrispondente capacità del cavo presente su quell'arco. Questa somma avrà un termine per ogni coppia di turbine (i, j), infatti, se considero la variabile x_{ij}^k essa assume valore 1 una sola volta per ogni i e j da 1 ad n. Garantito così che l'energia elettrica venga convogliata in qualche modo

lungo la rete, se l'energia decidesse di muoversi lungo un arco (i, j) allora, in virtù di tale vincolo sull'arco (i, j), avremo provveduto ad installare sufficiente capacità. Infatti, se f_{ij} è maggiore di 0 allora il modello impone di mettere le x a valori tali da fare in modo di supportare una capacità almeno pari al flusso. Di conseguenza, se le x non possono restare a 0, la necessità di installare dei cavi pone le y a 1, così da supportare la struttura topologica descritta. A questo livello, potremmo installare più collegamenti con diversi tipi di cavi tra due turbine, ma il vincolo che la somma delle x sia uguale a y fa si che la soluzione possa selezionare un solo tipo di cavo per ogni collegamento. Nel modello costruito, inoltre, non viene esclusa né la possibilità di poter costruire un collegamento non necessario in cui non passa corrente, (e selezionarci un cavo costosissimo) né la creazione di cicli. Per evitare queste situazioni e per concludere il modello, definisco una funzione obiettivo (1.1). Essa è legata solo al costo dei cavi, e richiede di minimizzare il costo totale del collegamento tra le turbine. La distanza tra la coppia di turbine (i, j), calcolabile grazie alle coordinate fornite in input, ci permette di conoscere la lunghezza necessaria per il cavo scelto e di conseguenza, attraverso il suo costo al metro, il costo totale del collegamento (i, j). La funzione obiettivo quindi minimizzerà la somma, per ogni i,j da 1 a n e per k da 1 a K, del prodotto tra x_{ij}^k e il costo di tale collegamento.

$$min \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{K} cost(k)\, dist(i, j)\, x_{ij}^k \tag{1.1}$$

$$y_{ij} = \begin{cases} 1 & \text{se costruisco l'arco } (i, j) \\ 0 & \text{altrimenti} \end{cases} \qquad \forall i, j = 1, \ldots, n \tag{1.2}$$

$$x_{ij}^k = \begin{cases} 1 & \text{se uso cavo di tipo } k \text{ sull'arco } (i, j) \quad \forall i, j = 1, \ldots, n \\ 0 & \text{altrimenti} \qquad\qquad\qquad\qquad\quad \forall k = 1, \ldots, K \end{cases} \tag{1.3}$$

$$f_{ij} \quad \begin{cases} \geq 0 & \text{se } i \neq j \\ = 0 & \text{altrimenti} \end{cases} \qquad \forall i, j = 1, \ldots, n \tag{1.4}$$

$$\sum_{j=1}^{n} y_{hj} = \begin{cases} 1 & \text{se } P_h \geq 0 \\ 0 & \text{se } P_h = -1 \end{cases} \qquad \forall h = 1, \ldots, n \tag{1.5}$$

$$\sum_{i=1}^{n} y_{ih} \leq C \quad \forall h : P_h = -1 \tag{1.6}$$

$$\sum_{j=1}^{n} f_{hj} = \sum_{i=1}^{n} f_{ih} + P_h \quad \forall h : P_h > 0 \tag{1.7}$$

$$y_{ij} = \sum_{k=1}^{K} x_{ij}^{k} \quad \forall i, j = 1, \ldots, n \tag{1.8}$$

$$\sum_{k=1}^{K} cap(k)\, x_{ij}^{k} \geq f_{ij} \quad \forall i, j = 1, \ldots, n \tag{1.9}$$

1.1.1 Modello con vincoli soft

La risoluzione del modello appena presentato può richiedere un tempo di calcolo molto lungo. Per velocizzarne la risoluzione utilizzeremo tecniche che verrano spiegate nei capitoli successivi. Per utilizzare questi metodi è necessario avere a disposizione, il prima possibile, di una soluzione *incumbent*, questo però avviene con forte ritardo in molte istanze che andremo ad analizzare. La mancanza di una possibile soluzione ammissibile, in tempi brevi, è dovuta alla presenza di vincoli troppo stringenti, a questo scopo è possibile modificare il modello. Una prima variante possibile è l'introduzione di una variabile $s \geq 0$ nel vincolo (1.6), che ci permetta di collegare più cavi alla substation rispetto al valore di C impostato in input. Così facendo abbiamo trasformato un vincolo rigido in un vincolo soft, dove si paga una penalità molto alta nella funzione obiettivo nel caso in cui $s > 0$. La soluzione ottima finale porterà quindi s a 0, essendo troppo costoso rispetto alle altre variabili, e infine risolverà il problema. Così facendo, avremo immediatamente a disposizione un *incumbent* potendo così applicare dei metodi chiamati euristici. Con un parametro in input, da noi chiamato relax, è possibile decidere se usare questo modello o quello base.

$$s \geq 0$$

$$min \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{K} cost(k)\, dist(i,j)\, x_{ij}^{k} + Ms \tag{1.10}$$

$$\sum_{i=1}^{n} y_{ih} \leq C + s \quad \forall h : P_h = -1 \tag{1.11}$$

Rispetto a quello precedente, quindi, cambierà la funzione obiettivo e il vincolo (1.6), mentre il resto del modello resterà invariato.
Una seconda alternativa al modello originale, ove $M \gg 0$ (nella nostra implementazione è stato posto a 10^9), considera la modifica del vincolo riguardante il flusso in uscita da ogni turbina. Ricordiamo che nel modello iniziale questo doveva essere uguale al flusso in entrata più la produzione della turbina in questione (1.7). Un rilassamento di questo vincolo prevede l'inserimento delle variabili $s_h \geq 0$, che permettano al flusso in uscita di essere inferiore rispetto a quello in entrata, consentendo così eventuali perdite di corrente lungo il percorso. Come nel rilassamento precedente (1.10), un eventuale di $s_h > 0$ nella soluzione finale deve provocare una penalità a livello di costo. Il modello, quindi, viene modificato in (1.1) e (1.7) nel seguente modo

$$s_h \geq 0 \quad \forall h = 1, \ldots, n$$

$$min \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{K} cost(k)\, dist(i,j)\, x_{ij}^{k} + M \sum_{h=1}^{n} s_h \tag{1.12}$$

$$\sum_{j=1}^{n} f_{hj} = \sum_{i=1}^{n} f_{ih} + P_h - s_h \quad \forall h : P_h > 0 \tag{1.13}$$

Se si volesse permettere un'eventuale soluzione in cui alcune turbine risultino completamente isolate, causando così un'intera perdita di corrente, dovremmo modificare, oltre a (1.1) e (1.7) come visto precedentemente, il vincolo (1.5) rendendolo una disuguaglianza nel caso in cui si stia prendendo in considerazione una turbina h con potenza positiva.

$$\sum_{j=1}^{n} y_{hj} \begin{cases} \leq 1 & \text{se } P_h \geq 0 \\ = 0 & \text{se } P_h = -1 \end{cases} \quad \forall h = 1, \ldots, n \tag{1.14}$$

Un ultimo rilassamento analizzato consiste nel permettere un'eventuale scelta del cavo meno costoso a discapito di una perdita di flusso nel caso in cui questo sia superiore alla capacità del collegamento installato. Per far ciò è necessario modificare il modello inserendo le variabili $s_{ij} \geq 0$ che penalizzino il costo della soluzione in caso di non ammissibilità dei vincoli originali. In questo caso il vincolo da modificare, oltre alla funzione obiettivo (1.1), è il (1.9)

$$s_{ij} \geq 0 \quad \forall i, j = 1, \ldots, n$$

$$min \sum_{i=1}^{n} \sum_{j=1}^{n} \sum_{k=1}^{K} cost(k)\, dist(i,j)\, x_{ij}^{k} + M \sum_{i=1}^{n} \sum_{j=1}^{n} s_{ij} \tag{1.15}$$

$$\sum_{k=1}^{K} cap(k)\, x_{ij}^{k} \geq f_{ij} - s_{ij} \quad \forall i, j = 1, \ldots, n \tag{1.16}$$

1.2 Implementazione base in CPLEX

CPLEX ha la possibilità di lavorare in due modi differenti: il primo attraverso comandi utilizzati da interattivo, il secondo mediante alcune routine di libreria. In questo capitolo ci occuperemo di alcuni dettagli implementativi svolti nella fase di programmazione, inerenti al secondo approccio. Il nostro scopo è quello di creare un programma che possa leggere dati di input forniti da riga di comando e da file di testo esterni, crei un modello matematico che risponda alle nostre esigenze, e infine lo ottimizzi utilizzando dei parametri caratteristici di CPLEX che ne migliorino l'efficenza nella risoluzione.

1.2.1 Struttura "instance"

"Instance" è una struttura dati che contiene tutte le informazioni di input o i parametri di CPLEX utili a risolvere il nostro problema e dove verrà salvata la soluzione. Quando nel *main* utilizziamo l'istruzione *instance inst*, chiediamo al sistema operativo di ritagliarci in memoria una struttura, contenente al suo interno tutte le variabili sopracitate, di tipo istanza il cui nome è *inst*. In questo modo creiamo la struttura ma al momento tutti i campi risultano essere non inizializzati. Inizialmente diamo a tutti questi dati un valore di default, per esempio pongo il *time limit* ad infinito utilizzando una costante di CPLEX, *CPX_INFBOUND*. Successivamente li riempiremo da riga di comando attraverso la funzione *parse_command_line* mentre altri da file esterni attraverso *read_input*.

1.2.2 Lettura della riga di comando

Per leggere i parametri in input, dati dalla linea di comando, utilizziamo la funzione *parse_command_line*. Definiamo i valori di default dei parametri, che andremo ad utiliz-

zare, entriamo in un loop dove leggiamo uno dopo l'altro gli argomenti forniti in input, se la *strcmp* applicata sulla stringa di comando i-esima ci restituisce 0 allora prendiamo il parametro successivo e lo inseriamo nell'opportuna variabile assegnatole. Se per esempio l'argomento i-esimo è il *time limit*, incrementiamo l'indice i di 1, passando all'argomento successivo scritto nella riga di comando, lo convertiamo da stringa a *floating point* con la funzione *atof* e lo memorizziamo in una variabile *timelimit* della nostra istanza creata in precedenza. Leggendo tutto ciò che ci serve dalla *command line* abbiamo memorizzato, nella variabile apposita della mia struttura dati, le informazioni utili a impostare dei parametri che utilizzeremo in seguito. È importante non definire le variabili in maniera globale ma racchiuderle in una struttura dati che passeremo alle varie funzioni che necessitano di alcuni parametri.

1.2.3 Lettura dei file di input

Read input è una funzione che ci permette di leggere dei file di input, nel nostro caso due. Il primo contiene le coordinate delle turbine e il tipo, identificato dalla potenza che essa può generare (1 nel caso di una classica turbina e -1 nel caso di una substation). Il secondo, invece, contiene le informazioni riguardanti i cavi a disposizione, essi sono descritti dalla loro capacità massima e dal prezzo al metro. I nomi di questi file vengono specificati da riga di comando, quindi sono letti dalla funzione *parse_command_line* e sono disponibili nell'istanza. Il *main*, quindi, dopo aver effettuato il parse della riga di comando e salvato i nomi dei file di input, chiama la funzione *read_input* e le passa l'indirizzo dell'istanza. Una volta letti i dati riempie con queste informazioni gli opportuni spazi precedentemente creati. Questa funzione apre un file alla volta e dopo aver svolto una prima lettura per scoprire la dimensione dello spazio necessario da allocare, lo richiede al sistema operativo attraverso la funzione *calloc*. Infine con una seconda lettura riempie effettivamente i campi riguardanti i dati trovati.

1.2.4 Ottimizzazione

Questa fase si svolge all'interno di *CABLEopt* e si suddivide in due fasi: la prima consiste nel chiamare la funzione *build_model* addetta alla costruzione del modello matematico, la seconda invece sarà la vera e propria fase di risoluzione. Inizialmente è necessario aprire uno spazio vuoto di CPLEX attraverso due sue funzioni caratteristiche:

- *CPXENVptr env = CPXopenCPLEX(&error)*

- *CPXLPptr lp = CPXcreateprob(env, &error, "CABLE")*

La prima definisce un environment di tipo *CPXENVptr* ovvero un puntatore di CPLEX ad un environment attraverso la funzione *CPXopenCPLEX*, se in questo momento avviene un errore viene catturato dalla variabile *error*. Questo environment successivamente viene passato ad un'altra funzione che si occupa di definire il modello lp, al momento ancora vuoto.

1.2.4.1 Costruzione del modello

L'operazione di costruzione del modello viene svolta da *build_model*. Ad essa è necessario passare l'istanza da cui prendere i dati di input, l'environment e lp, ovvero il modello vuoto che poi andrà a riempire.
Per creare una nuova variabile in CPLEX è necessario usare la funzione *CPXnewcols* che aggiunge una colonna al nostro modello. Per creare, per esempio, la variabile y_{ij} (1.2) è necessario ciclare su tutte le coppie di turbine (i, j), per i e j compresi tra 0 e il numero di turbine -1, definire il nome da assegnarle e infine chiamare la funzione *CPXnewcols* che ad ogni iterazione ne aggiunge una, caratterizzata dal costo che ha nella funzione obiettivo, lower e upper bound, tipo (per esempio binaria o continua) e nome che le è stato assegnato. Dopo aver popolato il modello con un tipo di variabile, faremo lo stesso lavoro per le altre.
Terminato l'inserimento delle variabili inseriamo i vincoli attraverso la funzione *CPXnewrows* che aggiunge una riga vuota al modello. Specifichiamo il senso del vincolo (E se equazione, G se maggiore o uguale e L se minore o uguale), ne diamo un nome e un *right and side*. Abbiamo creato così una riga vuota nel modello. Per cambiare i coefficienti di questo vincolo utilizziamo un ciclo che ci modifica quelli diversi da 0.
Finita questa fase è spesso utile stampare il modello con l'istruzione *CPXwriteprob*.

1.2.4.2 Risoluzione

La fase di risoluzione del modello matematico può essere svolta da interattivo, dopo aver fatto leggere il file lp creato con la funzione *CPXwriteprob*, tramite il comando *opt* da riga di comando. Una volta inviata la richiesta di ottimizzazione, il controllo passa a CPLEX che ci fornisce dei log a video e una volta trovata la soluzione terminerà. La fase di risoluzione solitamente ha una durata molto lunga, per questo motivo è spesso utile

ricorrere all'utilizzo di algoritmi che possono facilitarne l'esecuzione.
Da programma si lancia l'ottimizzazione con la funzione *CPXmipopt*. Terminata l'ottimizzazione dentro l'environment è presente la soluzione ottima. È infine consigliato di deallocare env e lp tramite la funzione *CPXfreeprob(env, &lp)* e *CPXcloseCPLEX(&env)* prima di terminare il programma.

1.2.4.3 Acquisizione della soluzione

Per poter estrarre la soluzione dall'environment è necessario utilizzare la funzione *CPXgetx*. Inizialmente è conveniente verificare l'esistenza della soluzione, in modo da evitare errori nel caso in cui questa non sia stata trovata. La funzione *CPXgetobjval* restituisce il valore della soluzione trovata, se questo valore non esiste (o è molto grande) significa che non è stato trovato nulla. Al contrario, se questo valore sembra essere accettabile chiameremo la *CPXgetx* in uscita dalla *CPXmipopt*. In caso di successo, è necessario dire quali sono i valori che ci interessano, quindi dobbiamo dare un puntatore ad array di double che memorizzi i valori della soluzione ottima. Bisogna comunicare il primo e l'ultimo indice di variabile che ci interessa ottenere. Nel nostro caso, essendo *ncols* il numero totale di colonne del nostro modello, dovremo allocarci con la funzione *calloc* un array di double della lunghezza necessaria e infine riempirlo con i valori della soluzione. Questo vettore avrà al suo interno i valori ottimi della y, della f e della x (in questo ordine per nostra decisione). Una volta estratte queste informazioni da CPLEX dovremo comunicare all'utente la soluzione che abbiamo trovato. La difficoltà a questo punto è a livello implementativo, infatti, per esempio, se volessi i valori ottimi delle y_{ij} sarebbe necessaria una funzione inversa di *ypos* (che ci mappava in una determinata posizione la variabile y_{ij}). Dovremmo scandire il vettore *solstar*, contenente la soluzione ottenuta precedentemente da CPLEX, saltare gli 0 e dove trovo qualcosa vicino al valore 1 (vista la possibilità di dati sporchi visto il formato double utilizzato) so che è rappresntato un arco scelto. Ci dovremmo quindi chiedere a quale (i, j) corrisponde quel valore data la sua posizione in memoria. In realtà questa funzione non è necessaria perché potremmo ottenere lo stesso risultato attraverso un ciclo *for* in cui consideriamo tutte le coppie (i, j) e definendoci la posizione con la funzione *ypos*.

2

Algoritmi euristici in CPLEX

Durante l'esecuzione, CPLEX ci mostra due particolari colonne nel suo *log* che rappresentano la forbice all'interno della quale si trova il valore della soluzione ottima. In un primo momento, CPLEX tenta di stimare un *lower bound*, inizialmente coincidente con il valore ottimo del rilassamento continuo. Successivamente, attraverso dei tagli, questa stima migliora. All'ottimo questo *lower bound* dovrebbe coincidere con la soluzione finale, ma questo risultato spesso è raggiungibile solo dopo un tempo molto lungo. Contemporaneamente CPLEX, con i suoi euristici o grazie ad una soluzione intera trovata, ha un incumbent che migliora. Il programma termina con la dimostrazione di ottimalità quando il gap fra l'*upper bound* (*incumbent*) e il *lower bound* (il migliore nei nodi aperti) coincidono. L'obiettivo è quello di ottenere questa coincidenza il prima possibile. Cambiando dei parametri il comportamento del software varia. Potremmo, per esempio, avere un peggior movimento del *lower bound*, lento nell'avvicinarsi all'ottimo, ma migliore dal punto di vista euristico, vista la produzione di una soluzione migliore. Posto che l'utente ci fornisca un limite di tempo entro il quale si aspetta di avere un risultato, questa potrebbe essere molto più soddisfacente essendo l'*incumbent* accettabile. La necessità, a volte, di migliorare il software sapendo che l'istanza non sarà risolvibile all'ottimo in tempo ragionevole, si traduce in una ricerca di una miglior soluzione possibile in tempi brevi.

Per far questo dobbiamo in qualche modo enfatizzare gli euristici nel run di CPLEX. Gli algoritmi euristici sono un particolare tipo di algoritmo spesso utilizzato per agevolare la

risoluzione di problemi molto complicati. Grazie al loro utilizzo è possibile, infatti, velocizzare la ricerca di una soluzione ottima (o molto buona) di un problema che altrimenti richiederebbe troppo tempo. Questi metodi permettono di ottenere risultati approssimati ma molto vicini a quelli ottimi. Questo tipo di approccio viene spesso utilizzato quando è necessario ottenere una soluzione in tempi brevi anche se non necessariamente ottimali. Un'azione possibile è quella di prendere la soluzione del rilassamento continuo, arrotondare le variabili, rendendola così intera ma molto probabilmente non ammissibile. Tuttavia se capitasse che lo fosse, allora verrebbe confrontata con l'*incumbent* e ci potrebbe essere un miglioramento. CPLEX di default non applica questi metodi in maniera estensiva, ma attraverso alcuni parametri potremmo usare per enfatizzare le fasi euristiche. In questo modo ci concentreremmo a migliorare l'*incumbent* e non il *lower bound*, che comunque solitamente resta abbastanza statico. In questo capitolo illustreremo alcuni degli algoritmi euristici più utilizzati.

2.1 RINS

RINS [3] è un euristico che esplora le possibili soluzioni vicine all'*incumbent* corrente del problema per cercare uno nuovo. L'esplorazione delle soluzioni viene effettuata risolvendo un altro MIP (subMIP) per un numero di nodi limitato. È stato sviluppato per CPLEX e di default non viene usato molto spesso perchè rallenta la fase di convergenza. Il parametro *CPX_PARAM_RINSHEUR* determina la frequenza con cui viene applicato. L'euristico tenta di migliorare immediatamente la miglior soluzione finora trovata, ma non può essere applicato finchè CPLEX non individua almeno una soluzione *incumbent*. Con valore -1 RINS non viene mai applicato, con valore di default 0 l'euristico viene applicato ad intervalli scelti automaticamente dal software; qualsiasi altro intero positivo fissa il numero di nodi dopo cui l'algoritmo viene applicato. RINS è di solito molto potente per individuare soluzioni buone.

2.2 Polishing

Il Polishing [4] è un algoritmo di post-processing di tipo genetico che lavora con una popolazione di soluzioni. Questa tecnica di pulizia agisce quando il run è finito, su una soluzione anche non ottima, attraverso una modifica della soluzione trovata, nel tentativo

di perfezionarla. In pratica vengono effettuati in automatico quei piccoli miglioramenti che un essere umano a volte riesce a fare "ad occhio". Questo meccanismo, a differenza di RINS, non viene applicato ad ogni nodo decisionale; ma solo quando il programma raggiunge una certa condizione di terminazione. Per applicare questo meccanismo dobbiamo dire a CPLEX dopo quanto tempo far azionare questo metodo. Per farlo ci basta impostare da programma attraverso la funzione *CPXsetdblparam* il parametro *CPXPARAM_MIP_PolishAfter_Time*.

3

Crossing

Una soluzione del nostro problema di ottimizzazione, potrebbe richiedere un'intersezione di due o più cavi. Ipotizzando che questi vengano stesi in maniera rettilinea, nel collegamento tra due turbine, dobbiamo fare in modo di evitare che queste situazioni si presentino. Per poter individuare eventuali incroci è necessario risolvere matematicamente un piccolo sistema di equazioni lineari che, dati i quattro punti dei due archi presi in esame, ci fornisce le coordinate del punto di un eventuale *crossing*. Prendiamo in considerazione i due segmenti $[P_1P_2]$ e $[P_3P_4]$, parametrizziamo ogni punto che va da P_1 a P_2 usando un parametro λ e la posizione di ogni punto che va da P_3 a P_4 con un secondo parametro μ. Questo vuol dire che, se esiste un punto appartenente ad entrambi i segmenti, esiste una soluzione, con λ e μ compresi tra 0 e 1, che rispetta il seguente sistema:

$$\begin{cases} X_1 + \lambda(X_2 - X_1) = X_3 + \mu(X_4 - X_3) & \lambda \in]0,1[\\ Y_1 + \lambda(Y_2 - Y_1) = Y_3 + \mu(Y_4 - Y_3) & \mu \in]0,1[\end{cases} \tag{3.1}$$

Se da questa risoluzione si scoprisse la presenza di archi sovrapposti potrebbe essere necessario riparare la soluzione. Per far ciò stabiliamo matematicamente una famiglia di vincoli che lo impedisca. Una prima possibilità è quella di imporre la seguente relazione:

$$y_{ab} + y_{cd} \leq 1 \quad \forall (a,b,c,d) : [P_a, P_b] \; incrocia \; [P_c, P_d] \tag{3.2}$$

Questa famiglia contiene potenzialmente n^4 vincoli, quindi una loro aggiunta nel modello lo renderebbe molto pesante. Il vincolo appena esposto risulta sicuramente corretto ma non necessariamente il migliore. In programmazione lineare intera esistono vincoli corretti ma più o meno efficaci in base alla riduzione che generano alla regione di ammissibilità del rilassamento continuo. Per migliorare (3.2) è possibile sfruttare l'irrilevanza dell'ordinamento tra due punti. Per far ciò consideriamo i quattro archi orientati tra le due coppie di punti (a, b) e (c, d) imponendo, al massimo, la scelta di uno:

$$y_{ab} + y_{ba} + y_{cd} + y_{dc} \leq 1 \quad \forall (a, b, c, d) : [P_a, P_b] \; incrocia \; [P_c, P_d] \tag{3.3}$$

Anche questo secondo vincolo risulta corretto, infatti, se la soluzione fosse intera, CPLEX genererebbe lo stesso risultato prodotto con il vincolo (3.2). Se considerassimo però il rilassamento continuo, il (3.3) è più forte del precedente. Ponendo infatti come obiettivo il voler tagliare, con un vincolo, il più possibile, è facile mostrare che la seconda condizione è più stringente della prima. Stiamo confrontando due disuguaglianze in cui la prima è del tipo $\alpha^T y \leq \alpha_0$ e la seconda $\beta^T y \leq \beta_0$ in cui α_0 e β_0 sono uguali e $\beta \geq \alpha$. Una soluzione frazionaria verrà tagliata più facilmente con il secondo vincolo piuttosto che con il primo: diremo così che (β, β_0) domina (α, α_0). Anche in questo caso abbiamo n^4 vincoli, ma essendo più forte non c'è motivo per non preferirla. Esiste un terzo possibile vincolo ancora più efficace, specifico di questa applicazione. Fissiamo un nodo c e inseriamo tutti gli archi uscenti da c, incrocianti il segmento (a, b), in un sottoinsieme da noi chiamato $Q(a, b, c)$, contenente anche (a, b) e (b, a). Considerando tutti gli archi contenuti in $Q(a, b, c)$ la soluzione potrà sceglierne al massimo uno. Il nostro modello, imponendo che non posso scegliere più di un arco in uscita da una turbina, non permetterà mai la scelta di due archi uscenti da c; per questo motivo, se ne provassimo a scegliere due, dovremmo scegliere (a, b) e uno uscente da c, questo non è possibile nel caso in cui questi si intersechino. La correttezza di questo vincolo quindi deriva dal nostro modello specifico.

$$Q(a, b, c) = \Big\{(a, b), (b, a)\Big\} \cup \Big\{(c, d) \in \delta^+(c) : [P_c, P_d] \; incrocia \; [P_a, P_b]\Big\}$$

$$\sum_{(i,j) \in Q(a,b,c)} y_{ij} \leq 1 \quad \forall a, b, c = 1, \ldots, n : a < b, c \notin \{a, b\} \tag{3.4}$$

Se confronto questo vincolo con i precedenti, avendo lo stesso termine noto ma fondamentalmente più termini nella sommatoria, è immediato notare che è spesso molto più efficace. Un ulteriore vantaggio riguarda l'ordine di grandezza, dove passiamo da un $O(n^4)$ dei casi precedenti ad un $O(n^3)$: miglioramento questo ancora più significativo nel nostro caso, essendo n abbastanza grande.

3.1 Lazy constraint

Definito il vincolo che risolverebbe il problema dei *crossing*, è necessario capire come gestirne il numero eccessivo, dato che potrebbe essere inefficiente inserirli direttamente nel modello rendendolo così molto pesante. Una soluzione deriva dall'utilizzo del *branch-and-cut*, nel quale è possibile generare al volo dei vincoli nel caso in cui la soluzione corrente li violi. CPLEX permette attraverso i *lazy constrait* di gestire situazioni di questo tipo: con questa metodologia i vincoli verranno messi nel pool del software e verranno generati ogni qualvolta avviene un operazione di separazione nei nodi dell'albero decisionale. CPLEX tramite la funzione *CPXaddlazyconstraints* permette il passaggio di vincoli da inserire nel pool. Quando CPLEX sta per aggiornare l'*incumbent* verifica se la soluzione intera trovata rispetta o meno i *lazy constraint*, se ne viola anche solo uno di questi, i vincoli corrispondenti vengono "rivitalizzati". Portati nel modello matematico, l'*incumbent* non viene aggiornato e il modello continua nel tentativo di trovare una nuova soluzione.

3.2 Metodo loop

La tecnica dei *lazy constraint*, nel nostro caso, non è sempre la più efficiente. Il suo utilizzo genera un numero di vincoli eccessivo rispetto a quelli che poi effettivamente verranno usati. Per ovviare a questo problema si possono usare due approcci differenti: il primo è chiamato metodo loop, mentre il secondo inserisce i vincoli *run time* durante il *branch-and-cut*. In questa sezione verrà preso in considerazione il primo approccio. Il metodo loop venne proposto negli anni settanta per il problema del commesso viaggiatore, ma venne considerato troppo rudimentale e fu abbandonato dopo la nascita del *branch-and-cut*. In realtà questo metodo, nel nostro caso, risulta essere molto efficace. Il suo meccanismo di risoluzione consiste nel risolvere all'ottimo il problema senza nessun vincolo di nocrossing, analizza la soluzione ottima trovata e, nel caso siano presenti uno

o più incroci, aggiungere i vincoli che ne vietino la scelta. Così facendo una soluzione non ammissibile già analizzata non si ripresenterà in un'eventuale esecuzione successiva. I vincoli vengono aggiunti in maniera statica al modello con la funzione *CPXnewrows*, sfruttando il numero ridotto di casi che effettivamente si possano presentare. Nel caso in cui non siano presenti crossing il metodo termina fornendo la soluzione ottenuta che sarà ottima e ammissibile. Ad ogni iterazione, se sono presenti delle sovrapposizioni di archi, verrà aggiunto un vincolo nuovo essendo impossibile che si sovrappongano due archi di cui ho già vietato il crossing con un vincolo in un'ottimizzazione precedente. E' garantita la terminazione dell'algoritmo anche se nel peggiore dei casi è richiesto un tempo molto grande. Il vantaggio per cui è preferibile utilizzare vincoli statici a quelli lazy è dovuto al fatto che ogni volta che chiamiamo la *mipopt* si sfrutta la fase di *preprocessing*. Questo vantaggio era annullato, negli anni settanta, a causa di tecnologie meno avanzate; per questo motivo l'utilizzo di questo metodo veniva ritenuto una pessima scelta di programmazione. Questa tecnica, però, richiede di effettuare numerose volte la fase di ottimizzazione, risolvendo da zero il problema senza crossing richiedendo ad ogni iterazione diversi minuti. Per ovviare a questo problema riduciamo il tempo di calcolo imponendo un piccolo time limit che ci permetta di uscire dalla *mipopt* in base a condizioni di terminazione anticipata, rinunciando così alla dimostrazione dell'ottimalità. Per rendere il più efficiente possibile questo metodo, inoltre, potremmo leggermente modificare questo schema di procedimento e permettere all'ottimizzazione, una volta ottenuta una soluzione che non presenta crossing, di continuare da dove aveva terminato nella fase precedente. Così facendo ricominciamo da capo solo quando è strettamente necessario, ovvero solo quando troviamo nuovi vincoli. La condizione d'uscita di questo ciclo, quindi, non è quella di aver trovato una soluzione senza incroci ma quella di aver ottenuto una soluzione ottima senza intersezioni tra i cavi.

3.3 Lazy Constraint Callback

Un metodo di risoluzione più efficiente rispetto ai precedenti riguarda l'utilizzo delle callback di CPLEX. Il nostro obiettivo è quello di aggiungere vincoli dentro al risolutore, e per far ciò quando chiamiamo la funzione *CPXmipopt* dobbiamo fare in modo che all'interno di quel run il codice, magari in modalità multithreading e quando necessario, debba essere in grado di generare i tagli che vietino i crossing. Nel metodo loop risolvevamo il problema con CPLEX per un determinato periodo di tempo, prendevamo il controllo,

aggiungevamo un taglio e poi ricominciavamo. Così facendo, avendo già sviluppato l'albero decisionale, ricominciare ogni volta risulterebbe uno spreco di tempo. Il metodo che invece viene discusso in questa sezione è stata una delle novità che ci ha permesso un gran miglioramento di prestazioni: esso prese il nome di *branch-and-cut*, ovvero genera tagli al volo lungo un unico albero decisionale.
Per poterne chiarire l'applicazione è importante chiarire il concetto di callback. Essa è un tipo di procedura usata da CPLEX che, di tanto in tanto, nei punti critici, chiama una funzione durante l'esecuzione dell'algoritmo risolutore principale. Nel nostro caso specifico, CPLEX chiama tantissime di queste funzioni e una in particolare viene chiamata quando si trova una soluzione che sta per aggiornare l'incumbent, cioè una soluzione ritenuta idonea, intera, ammissibile per i vincoli al momento presenti nel modello. Il risolutore dovrebbe aggiornare l'incumbent ma prima di farlo chiama una nostra funzione definita tramite *CPXsetlazyconstraintcallbackfunc*. Per far ciò è necessario catturare questa chiamata, che CPLEX farebbe a vuoto, tramite un operazione definita installazione della callback, in questo modo la *mipopt* chiamerà la callback installata quando necessario. E' possibile disinstallare una callback se richiesto. E' necessario notare che CPLEX, facendo un'operazione di preprocessing, potrebbe accorgersi che due variabili appaiono uguali e usarne una sola nella risoluzione del problema duplicandola nuovamente nella soluzione finale. Un'operazione del genere, detta di aggregazione delle variabili, fa sì che le variabili all'interno di CPLEX non siano le stesse di quelle fornite in input. Per fare in modo di poter interagire con il software, è quindi necessario specificare l'utilizzo dei nomi usati. Per far ciò, bisogna cambiare un parametro mettendo ad off il *CPX_PARAM_MIPCBREDLP*. Un secondo punto delicato riguarda l'utilizzo della programmazione multithread. Di default, nel momento in cui installiamo la callback CPLEX va single thread per evitare problemi di conflitto, ma nel caso fosse richiesto è possibile impostare il parametro *CPX_PARAM_THREADS* passando il valore corretto di thread da dover utilizzare, assicurandomi di rendere la funzione chiamata thread safe. Terminata l'installazione devo provvedere a definire il metodo da chiamare. I parametri richiesti da questa funzione non sono modificabili e sono un puntatore costante all'environment e alcuni puntatori a strutture dati. Tra queste strutture assumono particolare importanza il *cbhandle* che punta alla nostra istanza e un altro che ci indica se all'interno della callback è successo qualcosa di interessante. Un terzo parametro, *wherefrom*, ha invece lo scopo di fornire l'informazione che indica il chiamante della funzione. Il codice *CPX_CALLBACK_DEFAULT* indica che all'interno del metodo chiamato non è avvenu-

to niente di importante, se invece vengono aggiunti dei vincoli provvederemo a modificare questo parametro con *CPX_CALLBACK_SET*.
Lo scopo della callback è quindi quello di prendere la soluzione intera che sta per aggiornare l'incumbent, tramite la funzione *CPXgetcallbacknodex* e analizzarla verificando se sono presenti o meno dei crossing. Si provvede quindi a chiamare il separatore, anche lui thread safe, che deve fornire i tagli da dover aggiungere. Nel caso siano presenti, i vincoli violati (cioè i crossing) vengono impediti con l'inserimento di un taglio tramite la funzione *CPXcutcallbackadd*. La proprietà caratteristica di questo nuovo vincolo è la sua validità globale, il *branch-and-cut*, infatti, ha proprio questa particolarità, ovvero il taglio generato in un nodo può essere riutilizzato in tutti gli altri dell'albero decisionale (se valido in generale). Teoricamente si potrebbe implementare un altro tipo di callback chiamata *Usercutcallback*, caratterizzata dall'uso di un metodo molto simile a quello appena trattato con la differenza che viene chiamata ad ogni nodo anche quando la soluzione è frazionaria. Questo metodo avrebbe il vantaggio di non dover aspettare soluzioni intere per tagliarle, anticipando così la generazione di questi vincoli. Lo svantaggio principale, invece, è la possibilità di generarne troppi appesantendo inutilmente il codice. In questa applicazione non approfondiremo quest'ultimo metodo.

4

Algoritmi Mateuristici

Gli algoritmi metaeuristici sono una famiglia di euristici caratterizzati da una tipologia specifica di funzionamento. Come già anticipato nel Capitolo 2, gli algoritmi euristici sono dei metodi che non ricercano l'ottimalità di una soluzione ma si accontentano di un risultato molto vicino ad esso, col beneficio di un esecuzione più breve. Il punto di partenza è una scatola chiusa, rappresentante un metodo esatto basato su MILP (*Mixed Integer Linear Programming*), che dato un input fornisce una soluzione ottima ma in un tempo di calcolo eccessivo. Prima degli anni duemila, per progettare un euristico si tentava di lavorare su questa scatola, scrivendoci all'interno del codice che permettesse a CPLEX di migliorare la soluzione *incumbent*. Dall'inizio del nuovo millennio l'approccio è cambiato, tenendo la scatola chiusa si è deciso di porre maggior attenzione alla modifica del modello rendendolo euristico. Modificando l'input da fornire al metodo ottimo si rende così l'esecuzione e la risoluzione più semplice e veloce. Questo nuovo modo matematico di pensare cambiò il nome di questa famiglia in mateuristici.

4.1 Hard fixing

L'Hard fixing venne proposto nei primi anni del 2000 ed è un euristico molto efficiente, probabilmente uno dei più semplici ed efficaci di questa famiglia. Data una soluzione del nostro problema, detta di riferimento, fissiamo alcune variabili scelte modificando il

modello e quindi possibili soluzioni future. La scelta degli archi da fissare avviene casualmente ma decidendone a priori una certa percentuale; per esempio, una possibile opzione potrebbe essere quella di mantenere in soluzione (cioè di fissare) il 50% degli archi scelti nella precedente. Il fissaggio provocherà una modifica del lower bound delle variabili che verrà posto uguale a 1, così facendo una soluzione futura dovrà per forza scegliere l'arco fissato. A questo punto una nuova risoluzione di CPLEX richiederà meno tempo rispetto a prima, essendo le possibili scelte ridotte, e molto più semplice. Nella nostra implementazione inseriamo nella scatola il metodo delle callback coi lazy constraint. La funzione *hard_fix* inizialmente ricerca una soluzione di riferimento e successivamente effettua un loop. In quest'ultimo, ad ogni iterazione si definiscono gli archi da fissare con la probabilità definita, si cambiano i bound con il metodo *CPXchgbds*, si chiama *CPXmipopt* che risolve il problema modificato, dandogli un time limit ridotto entro il quale terminare la sua esecuzione, e infine se la soluzione ottenuta è migliore della precedente viene aggiornata quella di riferimento; dopo aver riportato i bound al valore originale, iteriamo nuovamente. Nel caso in cui si ottenga una soluzione peggiore, viene mantenuta quella vecchia come riferimento e fissiamo degli archi differenti: essendo la scelta casuale, ci aspettiamo di trovare una soluzione diversa nel run successivo.
Matematicamente quello che stiamo facendo è fissare un intorno della soluzione di riferimento entro il quale ricercare una soluzione migliore: maggiore sarà il numero di archi fissati, e più stretto sarà l'intorno. Dal punto di vista pratico, un'alternativa più efficiente prevede di cambiare la percentuale durante l'esecuzione, evitando così di perdere troppo tempo su una soluzione iniziale pessima o di ricercare inutilmente in un intorno grande di una soluzione quasi ottima. Un altro modo per velocizzare ancora di più questo metodo consiste nel modificare a 0 l'upper bound degli archi crossanti a quelli fissati, dato che questi archi non verranno mai scelti nella soluzione (così facendo, restringiamo ulteriormente il campo di ricerca di una possibile soluzione migliore a quella corrente). Come nel caso dei lower bound, è importante ricordarsi di riportare a 1 gli upper bound prima di ripetere il ciclo successivo.
Un'ulteriore possibilità nella scelta degli archi da fissare, specifica per la nostra applicazione, consiste nel suddividere l'area di copertura in settori identificati dall'angolo formato rispetto alla posizione della substation e scegliendo in maniera casuale un settore, si veda [2] per la descrizione di questa idea. Ad ogni iterazione, marchiamo le turbine al suo interno e fissiamo tutti gli archi entranti o uscenti da esse. Viceversa, potremmo decidere di ottimizzare solo ciò che è all'interno del settore e fissare quindi tutti gli archi apparte-

nenti a turbine esterne ad esso. A livello pratico, la marcatura delle turbine si traduce in una trasformazione delle coordinate geografiche in polari rispetto alla posizione della substation.

4.2 Local branching

Il Local branching [5] (o soft fixing) è una tecnica simile a quella appena descritta, che a differenza dell'Hard fixing, permette di automatizzare la scelta degli archi da fissare attraverso la creazione di un unico vincolo aggiunto al modello. Date una soluzione generica y ed una di riferimento y^{RIF}, rappresentate da un vettore di 0 e 1, è possibile calcolare la distanza di Hamming tra i due vettori binari, ovvero il numero di *flip* necessari per rendere uno uguale all'altro. Il vincolo che andremo ad inserire consiste proprio nel limitare il numero di possibili *flip* ad un valore arbitrario K:

$$\sum_{(ij):y_{ij}^{RIF}=0} y_{ij} + \sum_{(ij):y_{ij}^{RIF}=1} (1 - y_{ij}) \leq K \tag{4.1}$$

In pratica stiamo definendo, data una soluzione di riferimento, un suo intorno dentro al quale sono presenti tutte le soluzioni con distanza di Hamming inferiore o uguale al K dato. Questa tecnica pone delle basi per affrontare problemi in maniera euristica con un approccio differente rispetto a quanto visto precedentemente. La versione da noi appena illustrata viene detta simmetrica perchè tutti i *flip*, da 0 a 1 e viceversa, vengono trattati allo stesso modo. Nel nostro caso specifico il numero di 1, all'interno di una soluzione, è fissato a $n-1$, essendo un'antiarborescenza con n nodi. Sfruttando questa proprietà, non è necessario effettuare il conteggio su entrambe le possibili variazioni, infatti il numero di mutazioni da 0 a 1 provocherà lo stesso numero di *flip* da 1 a 0. Per tale ragione, è possibile introdurre una versione, detta asimmetrica, nel quale compare solo il secondo termine di (4.1)

$$\sum_{(ij):y_{ij}^{RIF}=1} (1 - y_{ij}) \leq K \tag{4.2}$$

Attraverso semplici passaggi matematici è possibile arrivare alla notazione (4.4)

$$\sum_{(ij):y_{ij}^{RIF}=1} y_{ij} \geq (\sum_{(ij):y_{ij}^{RIF}=1} 1) - K \tag{4.3}$$

$$\sum_{(ij):y_{ij}^{RIF}=1} y_{ij} \geq n - 1 - K \tag{4.4}$$

Tale vincolo, dato un K piccolo, si traduce nel fissaggio di una buona parte degli archi della soluzione di riferimento.
Come anticipato, il concetto è simile al metodo Hard fixing a differenza del fatto che lasciamo a CPLEX il compito della scelta, che viene effettuata attraverso il calcolo della distanza di Hamming e alla definizione di un intorno entro il quale ricercare una nuova soluzione. Da notare che dal punto di vista pratico, il metodo asimmetrico produce un numero di coefficienti diversi da zero molto inferiore rispetto a quello simmetrico, rendendo meno pesante il modello matematico e non rallentandone la risoluzione.
A livello pratico ricerchiamo inizialmente una soluzione di riferimento e successivamente all'interno di un ciclo, ad ogni iterazione, aggiungiamo il vincolo al modello. Dopo aver ottenuto la soluzione ottima locale nell'intorno di y^{RIF} di raggio K, iteriamo nuovamente modificando la soluzione di riferimento nel caso in cui questa sia migliore della precedente presa in considerazione, altrimenti incrementiamo K andando a ricercare in un intorno più ampio una soluzione che ci permetta di migliorare la precedente y^{RIF}. Nel caso in cui K inizi ad essere troppo grande, ovvero all'aumentare dell'intorno in considerazione, l'efficacia di questo metodo diminuisce e diventa quindi preferibile utilizzare una tecnica differente per concludere l'ottimizzazione del problema.
Un procedimento simile a quello appena descritto è il metodo RINS, come già accennato nel Capitolo 2.1 è implementato dentro CPLEX e ha come idea principale un fissaggio, non random ma per confronto, degli archi della soluzione. In questo caso, date una soluzione di riferimento e una generica, le due vengono confrontate e si fissano gli archi in comune. Idealmente viene creato un intorno ellittico in cui sono contenute le due soluzioni prese in considerazione e tutte quelle poco distanti da loro. CPLEX sceglie come soluzioni da confrontare quella ottima del rilassamento continuo al nodo e l'*incumbent*: nonostante una non sia intera e l'altra non ottima, il funzionamento del metodo non viene assolutamente compromesso.

5

Confronto algoritmi

In questo capitolo verificheremo, quantificando la bontà di un algoritmo, quanto esso risulti efficiente. Immaginando che il nostro obiettivo sia, dato un time limit, quello di vedere quale algoritmo funziona meglio, possiamo decidere di analizzarli da due punti di vista differenti. Il primo consiste nel dare un time limit breve e verificare quanto buona è la soluzione finale trovata, confrontando gli algoritmi dal punto di vista euristico. Il secondo, invece, prevede un time limit molto più lungo e ricerca quello che risolve all'ottimo il maggior numero di istanze entro il limite temporale predefinito. In una prima fase analizzeremo l'algoritmo da noi implementato al variare di alcuni parametri, per esempio tenendo attivo o meno l'opzione RINS, il polishing finale oppure usando il modello con *slack* o quello base. Così facendo avremo a disposizione vari algoritmi da confrontare: questa è la classica operazione effettuata in pratica per individuare quale algoritmo funziona meglio in un specifico campo di applicazione. Nel nostro caso, effettueremo questa analisi sul *test bed* della wind farm reso disponibile da [2], composto da istanze più o meno complicate con un determinato numero di turbine e tipi di cavi. In ognuna, inoltre, è presente una substation con una capacità indicante il numero massimo di cavi che può supportare in ingresso.

number	wind farm	cable set
01	wf01	wf01_cb01_capex
02		wf01_cb01
03		wf01_cb02_capex
04		wf01_cb02
05		wf01_cb05_capex
06		wf01_cb05
07	wf02	wf02_cb01_capex
08		wf02_cb01
09		wf02_cb02_capex
10		wf02_cb02
12		wf02_cb04_capex
13		wf02_cb04
14		wf02_cb05_capex
15		wf02_cb05
16	wf03	wf03_cb03_capex
17		wf03_cb03
18		wf03_cb04_capex
19		wf03_cb04
20	wf04	wf04_cb01_capex
21		wf04_cb01
26	wf05	wf05_cb04_capex
27		wf05_cb04
28		wf05_cb05_capex
29		wf05_cb05

Tabella 5.1: Test bed utilizzato

name	**site**	**turbine type**	**no. of turbines**	**C**	**allowed cables**
wf01	Horns Rev 1	Vestas 80-2MW	80	10	cb01-cb02-cb05
wf02	Kentish Flats	Vestas 90-3MW	30	∞	cb01-cb02-cb04-cb05
wf03	Ormonde	Senvion 5MW	30	4	cb03-cb04
wf04	Dan Tysk	Siemens 3.6MW	80	10	cb01
wf05	Thanet	Vestas 90-3MW	100	10	cb04-cb05

Tabella 5.2: Capacità della substation in base al set di turbine utilizzate, dati presi da [2]

Per eseguire questo confronto, eseguiamo il programma e al termine del run prendiamo delle statistiche che riportano per esempio le informazioni riguardo a quanto buona sia la soluzione finale in termini di costo, il tempo di calcolo impiegato nell'ottimizzazione, il valore del *lower bound* finale e la percentuale di gap. Per agevolare l'operazione di raccolta delle statistiche è conveniente stampare alla fine del run una riga riportante tutte le informazioni a noi necessarie (all'inizio della quale mettiamo la parola chiave STAT). Ogni run produrrà, quindi, delle statistiche di quell'esecuzione caratterizzate anche dai rispettivi parametri utilizzati. Questi dati li importiamo in un file di testo e attraverso il comando *grep* prendiamo tutti i file che contengono la parola STAT e li inseriamo in una tabella, dove sarà più semplice confrontarli. Per velocizzare il tutto, automatizzeremo il meccanismo con l'utilizzo di un programma batch. Sarà ora immediato individuare quali algoritmi forniscono una soluzione migliore in tempo più breve.

5.1 Random seed

In un run del software, lavorando con il *branch-and-cut*, svilupperemo un albero decisionale. La stessa versione di CPLEX, lanciata su macchine diverse, può avere tempi di calcolo molto variabili, questo prende il nome di *performance variability*. Il fatto che i tempi d'esecuzione siano diversi su macchine differenti non porta grande stupore, ma che la stessa macchina in una partizione Windows e in una Unix possa avere tempi molto diversi porta grande interesse alla ricerca, dove la comprensione di come questo possa essere possibile sembra inspiegabile. Per questo motivo, dal 2000 alcuni gruppi di ricerca, nel tentativo di capire cosa avveniva, scoprirono che il problema era causato dalla determinazione dell'albero decisionale e dalla scelta della variabile di branching. Per semplicità, ipotizziamo che CPLEX abbia un suo criterio e decida di ramificare sulla variabile con parte frazionaria più vicina a 0.5. Ipotizzando che ne siano presenti più di una, l'algoritmo sceglierà, con una sua regola, la prima. Se così fosse, lanciandolo su macchine diverse, la variabile di branching al nodo radice scelta dovrebbe essere la stessa, in realtà questo non avviene. Infatti, cambiando il *floating point* della macchina succede che la stessa soluzione frazionaria venga memorizzata internamente con piccoli errori di percezione. Per questo motivo, quando l'algoritmo va in esecuzione, la scelta potrebbe ricadere su una variabile differente leggermente distaccata dalla precedente (0.499999 invece che 0.5). Essendo impossibile avere variabili di branching identiche, su macchine differenti, avremo tempi di calcolo completamente discordi. Un run fortunato potrebbe generare

un centinaio di nodi e terminare, mentre uno sfortunato ne potrebbe generare milioni. Questa è una caratteristica legata al metodo di risoluzione e non al software utilizzato. Il *branch-and-cut* risulta essere un sistema caotico. Per i motivi appena esposti, nel confronto tra due algoritmi, potremmo avere una differenza in tempi di calcolo enormi, dovuta a piccole variazioni effettuate nelle scelte di ramificazione dell'albero decisionale. Diventa, quindi, insensato giudicare un algoritmo più efficiente di un altro essendo una statistica magari legata alla macchina stessa e non al metodo utilizzato; la *performance variability* potrebbe così invalidare tutte le analisi statistiche svolte. Un meccanismo di stabilizzazione, inventato anni fa, mette a disposizione di CPLEX quello che viene chiamato *random seed*. Esso è un parametro che stabilisce una sequenza di numeri casuali. Cambiando il random seed si cambiano lievemente le politiche di scelta delle variabili di branching. Cambiando il seme random di CPLEX, senza cambiare l'algoritmo potremmo vedere dei run più o meno fortunati. Anch'esso, come gli altri parametri, può essere cambiato da interattivo o attraverso la funzione *CPXsetintparam*. In questo modo potremmo provare il nostro algoritmo con semi diversi verificando che l'andamento del run può essere a volte completamente diverso. Variando il random seed potremo quindi raccogliere più statistiche sulla stessa istanza, avendo così qualcosa di molto più significativo rispetto ai dati raccolti in precedenza dove la casualità rendeva i tempi di calcolo non affidabili.

5.2 Test svolti

In questo paragrafo riportiamo le analisi effettuate e i metodi implementati; per semplicità di confronto sono state scelte quattro istanze difficili in quanto non risolte all'ottimo nemmeno nell'articolo da noi preso in riferimento [2]. In un secondo momento è stato invece analizzato il comportamento delle tecniche, da noi usate, su un'istanza più semplice e risolta all'ottimo. I test sono stati svolti sul cluster di calcolo dipartimentale Blade [9] e i relativi dati raccolti sono stati analizzati e graficati tramite l'utilizzo di Microsoft Excel e Gnuplot [10]. Gli algoritmi implementati e testati con il supporto di CPLEX vengono riportati in Tabella 5.3.

Metodo	Descrizione
Lazy constraint	I vincoli di nocrossing vengono inizialmente messi tutti nel pool del software, e sono automaticamente utilizzati ogni volta che avviene un'operazione di separazione nei nodi dell'albero decisionale.
Metodo loop 15s - 30s	Risolve il problema senza nessun vincolo di nocrossing, analizza la soluzione ottima trovata e nel caso siano presenti uno o piu incroci aggiunge i vincoli che ne vietino la scelta. Il tempo a disposizione per ogni singola iterazione risolutiva è stato settato a 15 e successivamente a 30 secondi.
Callback	Utilizzo delle callback di CPLEX con aggiunta di vincoli lazy ad ogni tentativo di aggiornamento dell'incumbent nel caso in cui questo non sia ammissibile a causa di presenza di incroci tra i cavi.
Hard fixing 50%	Fissaggio della metà dei cavi attraverso una scelta casuale.
Hard fixing gap	Fissaggio di un numero casuale di cavi legato al gap tra il costo della soluzione attuale e il lower bound trovato nella soluzione di riferimento di partenza.
Hard fixing in sector - out sector	Fissaggio di tutti i cavi presenti all'interno o all'esterno di un settore composto dalle turbine posizionate in una certa zona formante un determinato angolo rispetto alla substation. Il settore viene scelto casualmente ad ogni iterazione dell'esecuzione; si veda [2].
Hard fixing local	Versione Hard fixing del Local branching, ovvero i fissaggi non avvengono con l'aggiunta di un vincolo al modello ma attraverso la modifica provvisoria del bound della variabile.
Local branching	Automatizzazione della scelta degli archi da fissare in un intorno, sempre più grande in caso di ottimo locale, della soluzione attuale. Nel caso in cui l'intorno superi una distanza di Hamming di 20 dalla soluzione di riferimento viene attivato il metodo Hard fixing sector che concluderà il run fino all'esaurimento del timelimit stabilito.

Tabella 5.3: Descrizione dei metodi implementati

Per ulteriori informazioni riguardo ai metodi implementati e al loro funzionamento si consulti il codice del programma sviluppato e il capitolo teorico in cui l'argomento viene approfondito in questa tesina. Per evitare i problemi citati nella sezione precedente è stato deciso di effettuare gli stessi test per cinque volte con semi random differenti. Per ogni run sono stati settati i parametri riferenti all'utilizzo di RINS, il tipo di rilassamento del modello usato e il timelimit desiderato. In particolare sono stati svolti i test con i seguenti valori riportati in Tabella 5.4.

Parametro	Valore	Descrizione
RINS	0, 10	Applica l'euristico RINS presente in CPLEX.
Relax	0, 1, 2, 3	Applica un tipo di rilassamento differente al modello matematico di partenza. Con il parametro settato a 0 si indica l'assenza di modifiche al modello originale, con 1 si permette un superamento di cavi in ingresso alla substation oltre la capacità prefissata; il rilassamento 2 riguarda un eventuale perdita di flusso di corrente, e infine il 3 permette un utilizzo di cavi di capacità inferiore a quello necessario
Timelimit	0, 30, 60, 90, 120, 150, 180, 240, 300, 360, 420, 500, 600	Impone un limite di tempo in secondi per l'esecuzione del metodo

Tabella 5.4: Descrizione dei parametri utilizzati

Dai primi risultati ottenuti abbiamo subito notato che con RINS impostato a 0 e senza l'utilizzo di rilassamenti, le soluzioni acquisite mostrano costi eccessivi in quanto inammissibili e per questo motivo non vengono riportati i relativi grafici. Vengono ora discussi inizialmente i risultati ottenuti per le istanze 1, 3, 5 e 26 e infine per la 19 dove le analisi effettuate si differenziano dalle precedenti, in quanto non si verifica più il costo della soluzione in funzione del tempo ma la velocità con cui la raggiunge, essendo risolta rapidamente all'ottimo. Per tali ragioni analizzeremo i risultati verificando quale dei metodi implementati risulti più rapido nel trovare e dimostrare l'ottimalità della soluzione, azzerando quindi il gap tra l'upper e il lower bound. Nelle istanze riportate di seguito, vengono riportati risultati per RINS settato a 10 per tutte, mentre nel caso dell'istanza 19 viene riportato anche il caso di RINS 0, riuscendo ad ottenere risultato apprezzabili.

5.2.1 Istanza 1

Vengono ora analizzati e confrontati i test svolti sull'istanza 1. Come spiegato precedentemente, questa risulta appartenere alla classe di istanze complicate da risolvere.

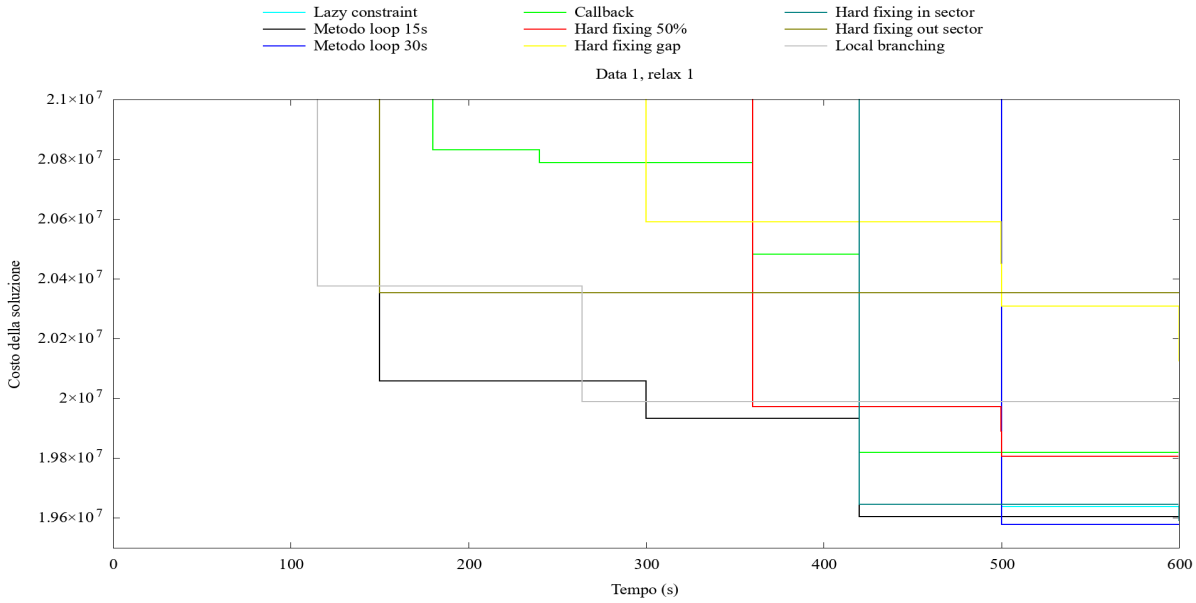

Figura 5.1: Analisi del costo della soluzione per l'istanza 1 con rilassamento di tipo 1

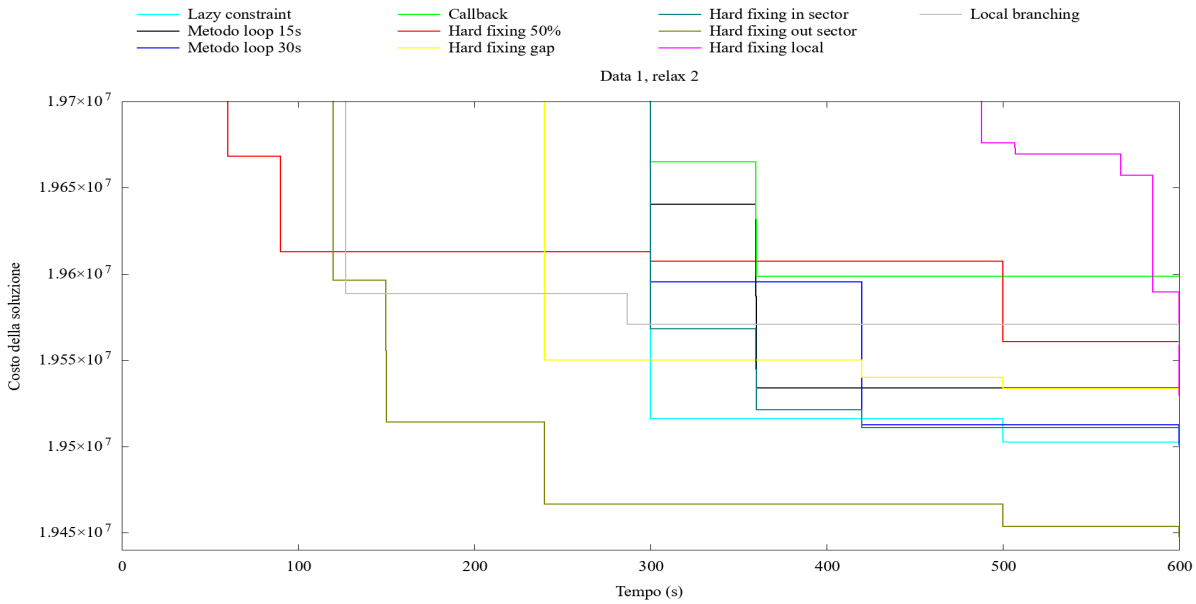

Figura 5.2: Analisi del costo della soluzione per l'istanza 1 con rilassamento di tipo 2

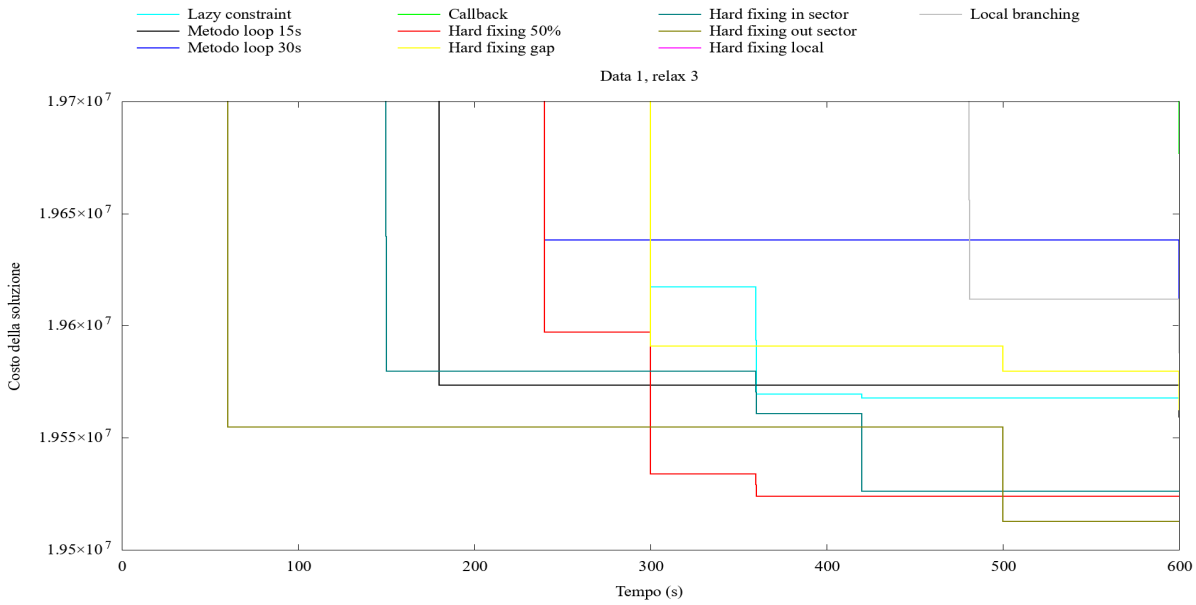

Figura 5.3: Analisi del costo della soluzione per l'istanza 1 con rilassamento di tipo 3

E' possibile notare dai grafici che, sulla stessa istanza, i metodi di rilassamento favoriscono metodi di risoluzione differenti. Nel tipo 1, il metodo che riesce ad ottenere nei 600 secondi di esecuzione un costo della soluzione minore è il metodo loop con durata 30 secondi. Negli altri due casi, invece, la tecnica più efficiente risulta essere quella che tenta di ottimizzare i collegamenti tra le turbine posizionate fuori dal settore di riferimento tramite l'hard fixing. Tale metodologia riesce ad ottenere il costo più basso per la funzione obiettivo del nostro modello, sia per il rilassamento di tipo 2 che per la tipologia 3.

5.2.2 Istanza 3

Per l'istanza 3, come nella precedente, vengono analizzati i test effettuati applicando al modello vari rilassamenti. In questo caso però, il tipo 1 non permette di ottenere risultati apprezzabili, in quanto il costo della funzione obiettivo resta molto elevato. Il costo, in questo caso, si presenta con vari ordini di grandezza maggiore rispetto all'ottimo rilevato negli altri casi e per questo motivo il grafico del metodo non viene riportato. Di seguito sono mostrati i grafici per il rilassamento di tipo 2 e 3.

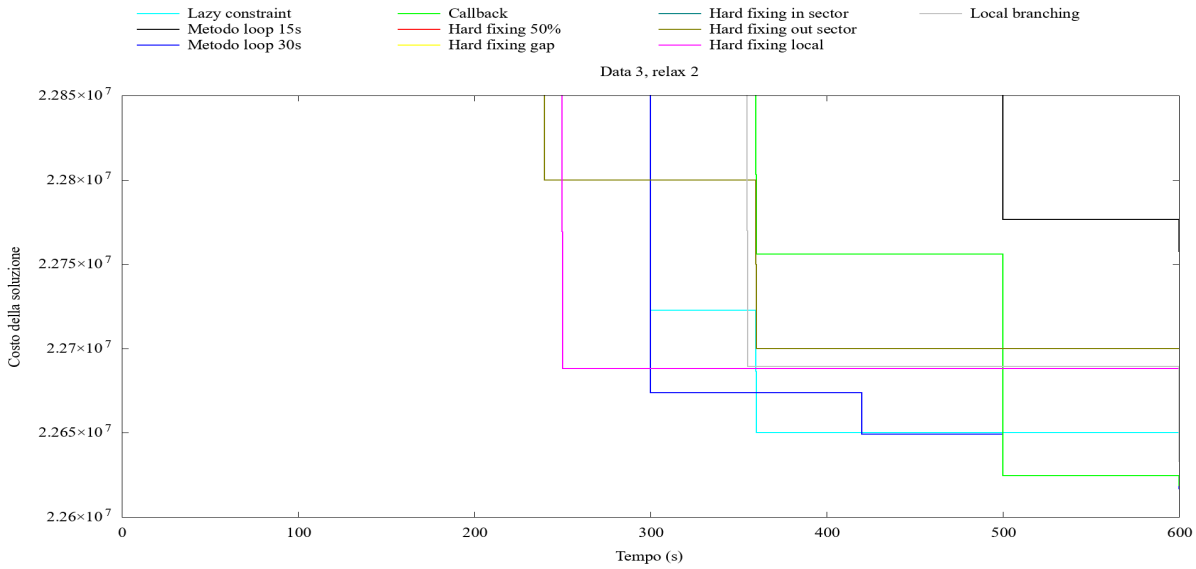

Figura 5.4: Analisi del costo della soluzione per l'istanza 3 con rilassamento di tipo 2

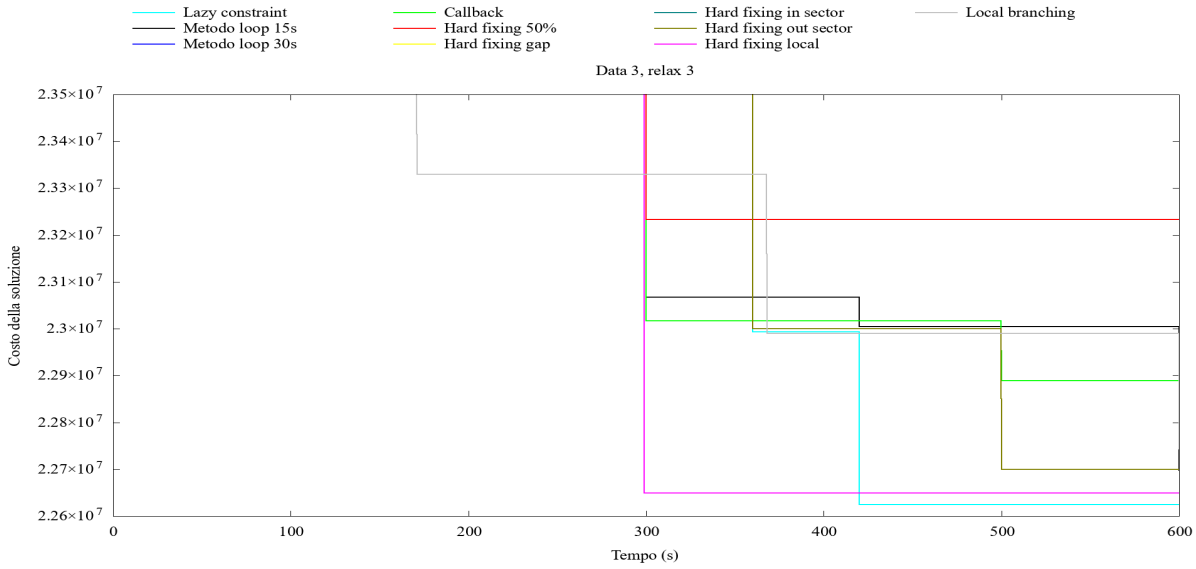

Figura 5.5: Analisi del costo della soluzione per l'istanza 3 con rilassamento di tipo 3

Nel metodo di risoluzione tramite il rilassamento di tipo 2, il metodo che riesce ad offrire il costo più basso per la nostra soluzione è quello che applica le callback. Nell'altro caso la tecnica di aggiunta dei lazy constraint fornisce il valore più basso per la funzione obiettivo. Questi dati differiscono rispetto al caso dell'istanza 1 dove, per esempio, il metodo delle callback si rivela il peggiore di quelli analizzati nel rilassamento 2.

5.2.3 Istanza 5

Nel caso dell'istanza 5, ci troviamo in una situazione simile alle precedenti. Nel caso del rilassamento 2, il metodo migliore risulta ancora quello delle callback, mentre per il rilassamento 3 torniamo al metodo dell'hard fixing applicato fuori dal settore di riferimento. Come per l'istanza 3, il rilassamento di tipo 1 non ha portato valori interessanti, per tale ragione non vengono graficati i risultati ottenuti.

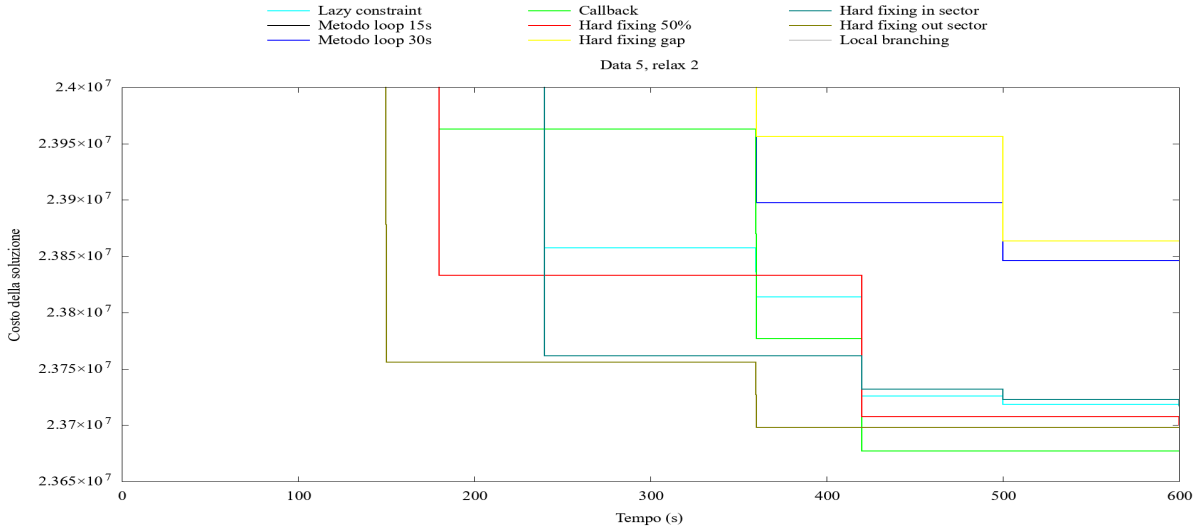

Figura 5.6: Analisi del costo della soluzione per l'istanza 5 con rilassamento di tipo 2

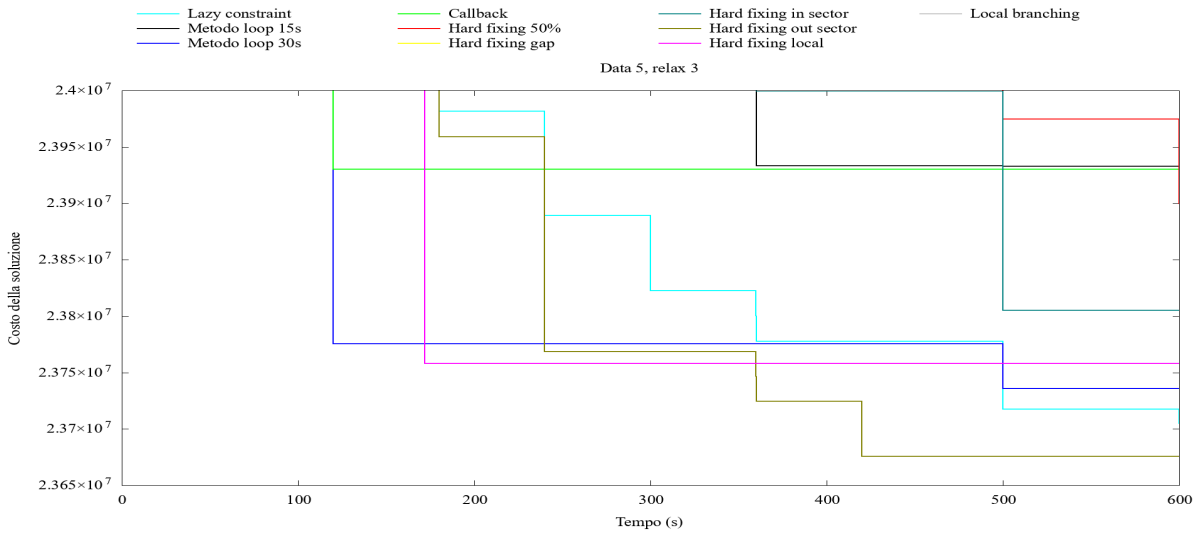

Figura 5.7: Analisi del costo della soluzione per l'istanza 5 con rilassamento di tipo 3

Le istanze finora analizzate, essendo molto simili strutturalmente, potrebbero far pensare ad un confronto ed una invarianza dei risultati ottenuti al variare dei metodi applicati, cosa che invece non avviene in quanto le prestazioni variano in modo netto in base all'istanza di applicazione.

5.2.4 Istanza 26

L'istanza 26 differisce rispetto alle altre in quanto la sua struttura presenta la substation nel mezzo delle turbine e non più al bordo del parco eolico. Come riportato nei grafici rappresentati in Figura 5.8 e 5.9, il metodo delle callback risulta essere il migliore sia nel rilassamento di tipo 2 che in quello 3. Nel caso della seconda tipologia è possibile ottenere prestazioni molto simili applicando il metodo loop con durata 30 secondi. Anche in questa istanza il rilassamento di tipo 1 non porta a risultati interessanti.

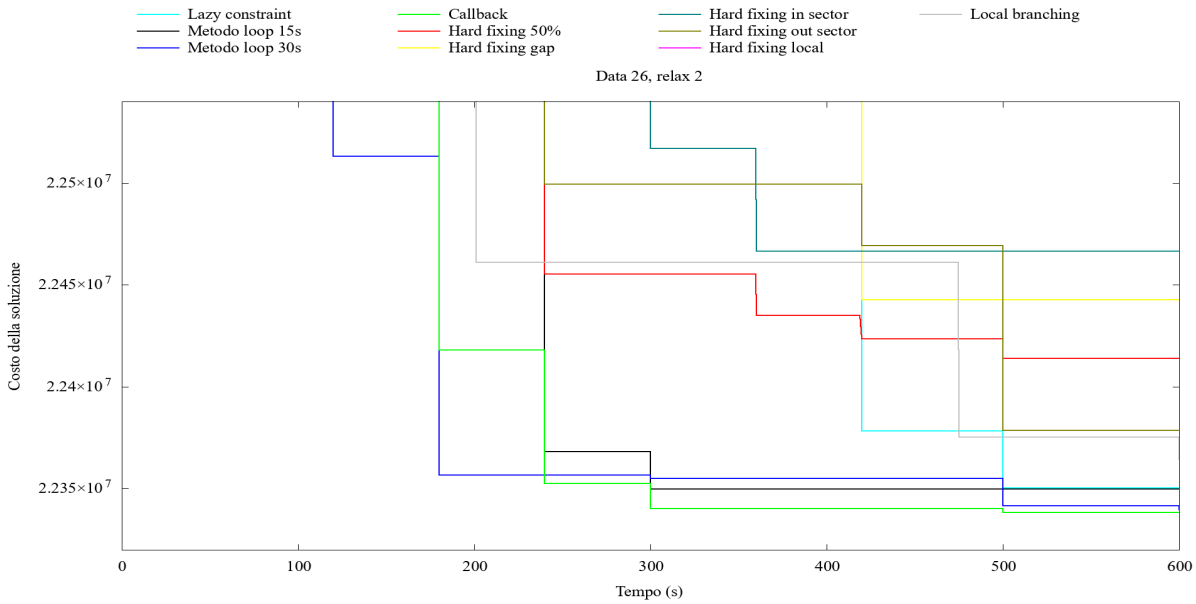

Figura 5.8: Analisi del costo della soluzione per l'istanza 26 con rilassamento di tipo 2

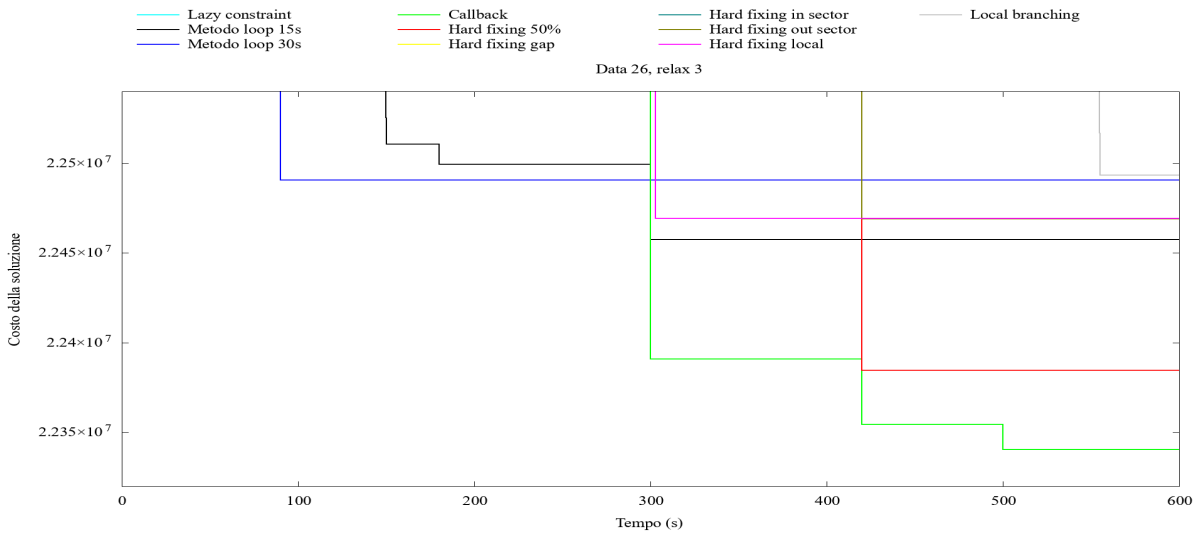

Figura 5.9: Analisi del costo della soluzione per l'istanza 26 con rilassamento di tipo 3

5.2.5 Istanza 19

Come precedentemente annunciato, sull'istanza 19 sono stati svolti dei test differenti in quanto risulta possibile la risoluzione all'ottimo in tempi ridotti. Essendo più semplice, si è notato che anche le esecuzioni svolte con RINS impostato a 0 hanno portato a buoni risultati. La semplicità di questa ottimizzazione è legata alla distribuzione spaziale delle turbine, che si presentano inoltre in numero più ridotto rispetto ai casi precedenti.

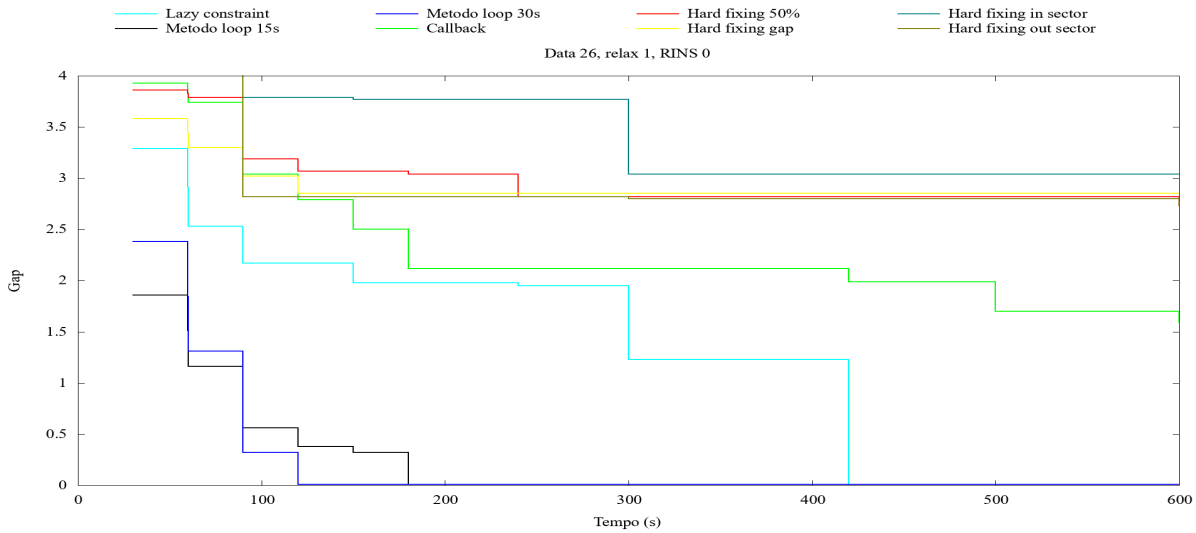

Figura 5.10: Analisi del costo della soluzione per l'istanza 19 con rilassamento di tipo 1 e RINS 0

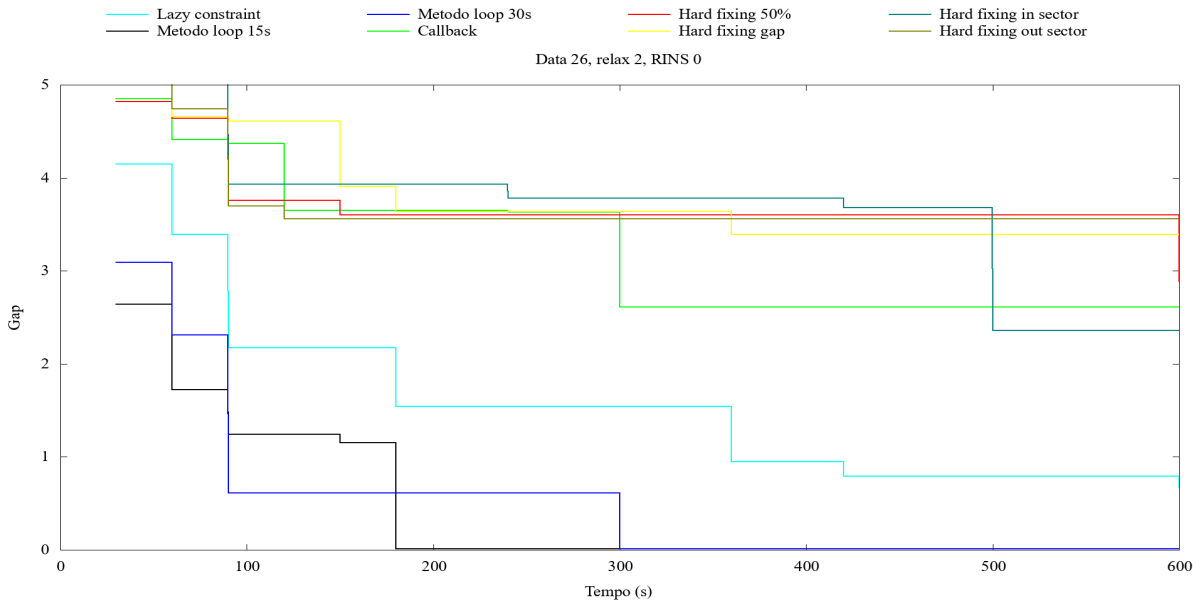

Figura 5.11: Analisi del costo della soluzione per l'istanza 19 con rilassamento di tipo 2 e RINS 0

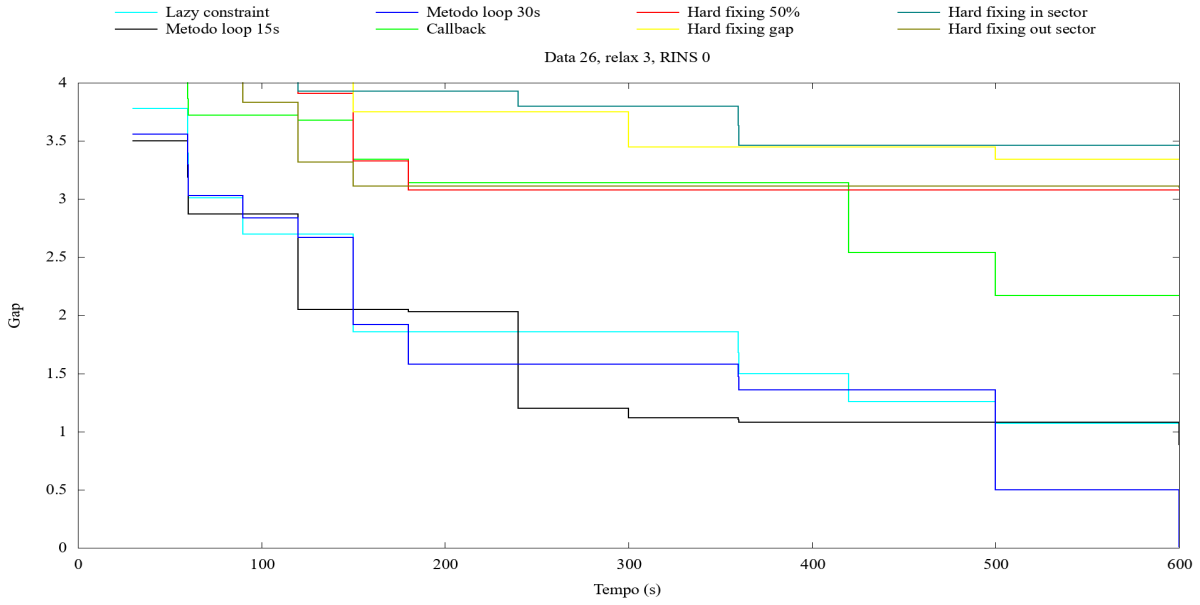

Figura 5.12: Analisi del costo della soluzione per l'istanza 19 con rilassamento di tipo 3 e RINS 0

Nel caso di RINS con valore 0, per tutti i rilassamenti riusciamo a risolvere all'ottimo il problema con il metodo loop di durata 30 secondi. Per quanto riguarda le prime due

tipologie otteniamo lo stesso risultato anche con il Loop da 15, e nel primo anche con il metodo dei lazy constraint.

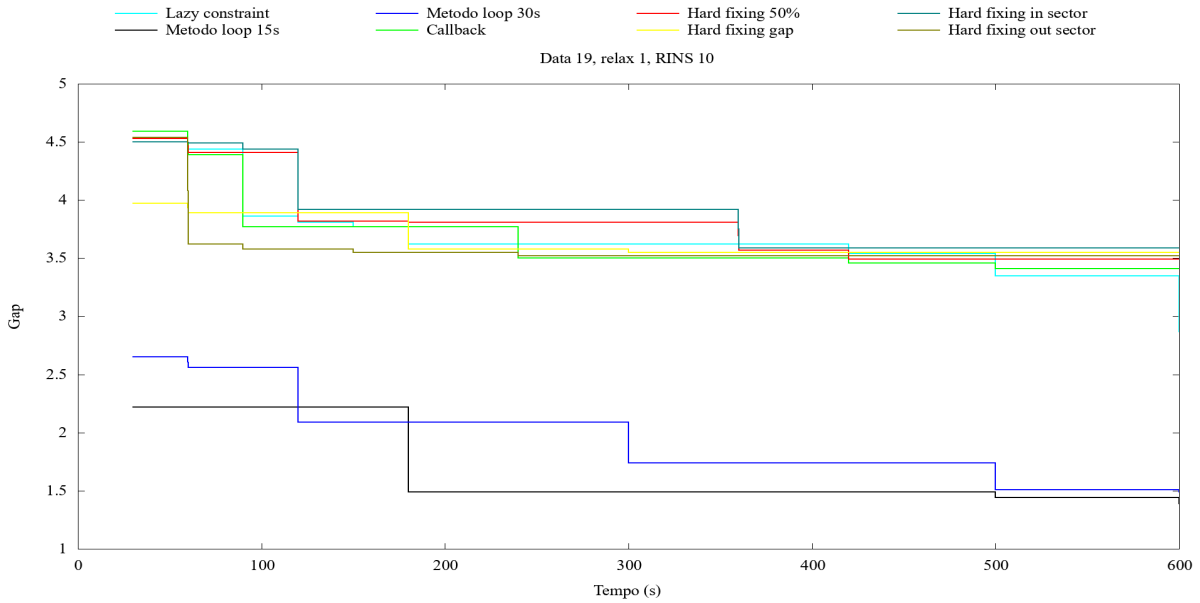

Figura 5.13: Analisi del costo della soluzione per l'istanza 19 con rilassamento di tipo 1 e RINS 10

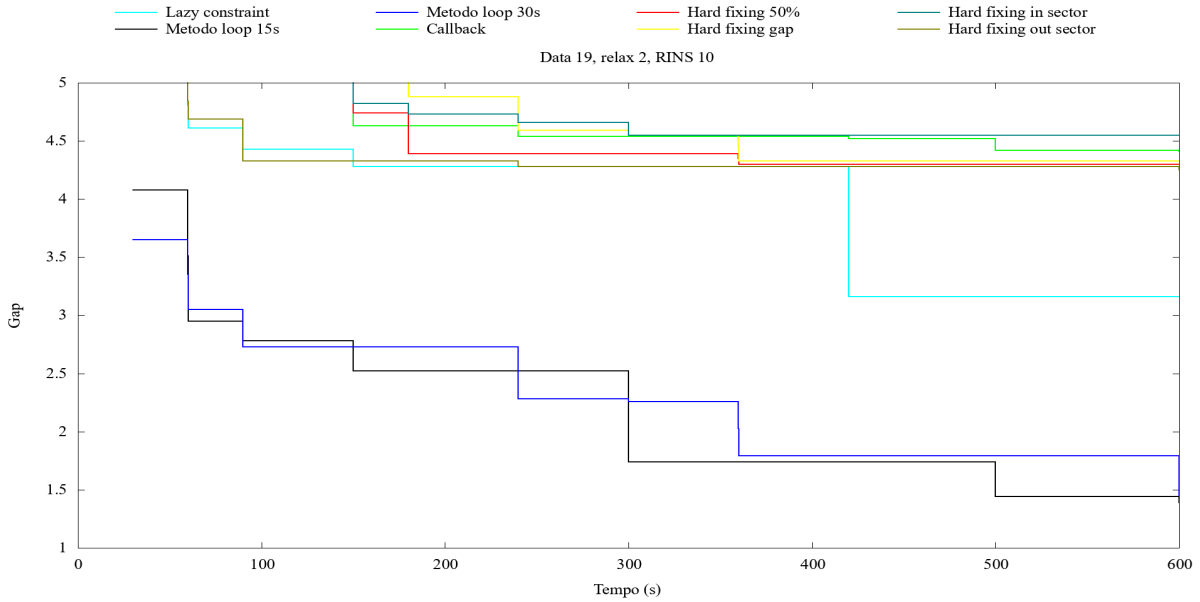

Figura 5.14: Analisi del costo della soluzione per l'istanza 19 con rilassamento di tipo 2 e RINS 10

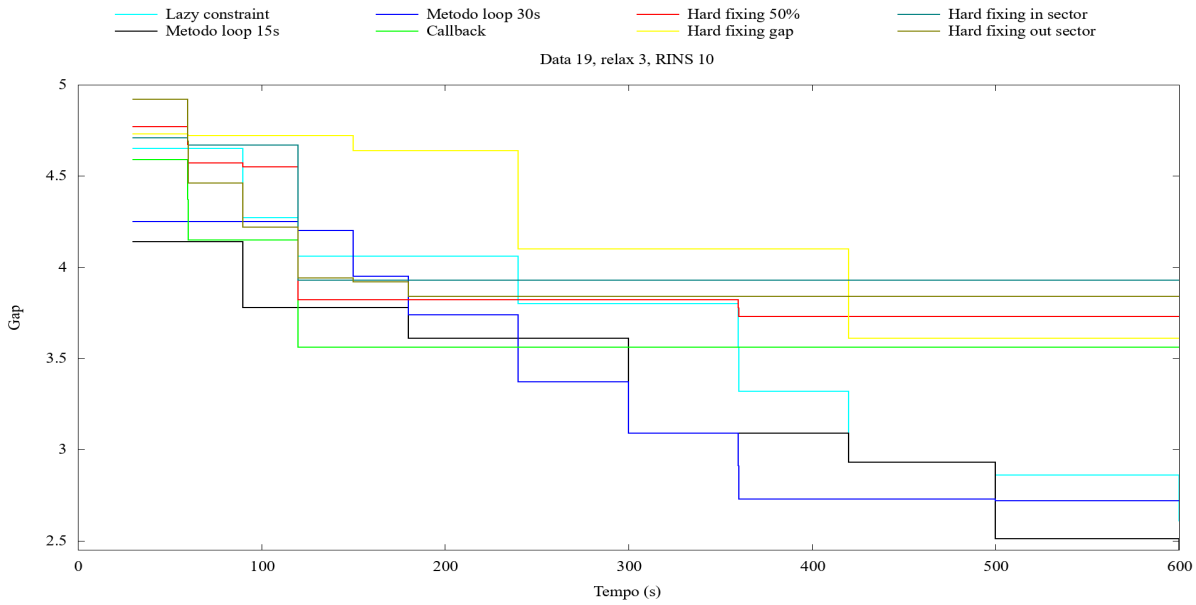

Figura 5.15: Analisi del costo della soluzione per l'istanza 19 con rilassamento di tipo 3 e RINS 10

Nel caso di RINS impostato a 10 non otteniamo invece soluzioni ottime in 600 secondi; nonostante ciò, è possibile notare delle analogie con i casi precedenti per quanto riguarda il comportamento dei vari metodi. I due metodi loop, sia di durata 15 che 30 secondi, risultano essere i migliori, seguiti sempre dal metodo dei lazy constraint, e per tale ragione questi due metodi si rivelano essere molto adatti ed efficaci nella risoluzione di questa istanza.

6

Ricerca di una soluzione euristica senza l'utilizzo di CPLEX

Analizzati gli algoritmi basati su CPLEX, in questo capitolo proveremo a riprodurre i risultati ottenuti senza l'utilizzo di questo software. Per far ciò è necessario affidarsi a dei metodi euristici che permettono di risolvere il problema senza la creazione esplicita di un modello matematico. Gli euristici possono essere classificati in due categorie: costruttivi e di raffinamento. La prima tipologia permette di ricercare una prima soluzione da cui partire, la seconda invece permette di migliorare la soluzione a propria disposizione.

6.1 Euristici costruttivi

Uno dei metodi costruttivi più famosi è il *greedy*. Questa politica permette di raggiungere una soluzione al problema in passi successivi nei quali ogni volta viene effettuata la scelta che si ritiene migliore, senza mettere mai in discussione decisioni passate. Questa tecnica non garantisce l'ottimalità della soluzione ma è spesso molto utile in fase costruttiva. Nel nostro caso specifico, un metodo *greedy* potrebbe prevedere di collegare tutte le turbine più vicine tra loro, saltando scelte che creerebbero cicli all'interno dell'albero. Una tecnica migliore, invece, prevede l'utilizzo dell'algoritmo di Dijkstra. Tale metodologia ci fornisce un albero di costo minimo come soluzione di partenza, anche se molto probabilmente non

ammissibile a causa della violazione del vincolo del numero di cavi in ingresso alla substation, oppure a causa di presenza di intersezione tra gli archi, oppure per la presenza di connessioni che richiederebbero dei cavi di capacità troppo elevata. Dal punto di vista implementativo creiamo un albero orientato nel quale, dato un nodo i, il suo successore viene etichettato come *suc[i]*; di default definiamo $suc[substation] = -1$. A questo punto, definita la struttura, è possibile calcolare il suo costo in base ai collegamenti utilizzati per supportare il flusso e alla loro lunghezza dovuta alla distanza tra le turbine collegate, imponendo una penalità molto elevata nel caso in cui siano presenti dei crossing oppure se il numero di cavi entranti nella substation sia superiore alla sua capacità di ingresso. Notando la possibile creazione di un albero in cui il flusso di un collegamento è maggiore alla capacità di ogni cavo a disposizione si è deciso, in un secondo momento, di inserire un'ulteriore penalità nel caso in cui sia presente una perdita di corrente lungo il cammino dalle turbine alla substation. Essendo la creazione di una soluzione molto rapida, randomizziamo l'algoritmo in modo tale da effettuare più iterazioni e ricercarne una migliore da poi raffinare in seguito. Una prima possibilità è chiamata *multi-start* e prevede di cambiare il punto di partenza della creazione dell'albero decisionale; la limitazione di questa tecnica è legata al numero di volte che può essere utilizzata, dato che il numero di differenti punti di partenza è il numero di nodi dell'albero. Una scelta migliore, chiamata *GRASP*, consiste nel salvarci, ad ogni passo, vari "migliori modi" di proseguire la ricerca, e invece di attuare ogni volta la scelta migliore forzare, ogni tanto, ed in maniera random, la soluzione a scegliere un arco tra le altre possibili alternative a disposizione. In questo modo siamo sicuri che in un numero elevato di iterazioni risulti praticamente impossibile ottenere due soluzioni identiche.

6.2 Euristici di raffinamento

Esistono varie tecniche di raffinamento, in particolare esiste una classe di algoritmi chiamati *Metaeuristici*. Data una soluzione da cui partire, queste tecniche tentano di migliorarla tramite un'operazione detta *mossa*, ovvero una regola che da una soluzione ne produce un'altra. Una possibile mossa consiste nel scegliere un arco, eliminarlo e sostituirlo con un altro. Per fare in modo che questo abbia senso, dobbiamo evitare di inserire il nuovo arco all'interno della componente connessa, altrimenti creeremmo un ciclo che ne penalizzerebbe il costo. Così facendo stiamo definendo un intorno della soluzione attuale e stiamo ricercando la migliore al suo interno. La caratteristica di questo intorno è che

tutte le soluzioni al suo interno sono a distanza *1-opt* da quella attuale, ovvero ad una distanza unitaria in base alla metrica utilizzata. E' necessario calcolare il costo di tutte le possibili soluzioni presenti nell'intorno e nel caso in cui ne venga trovata una migliore la salviamo. Il passo successivo consiste nel centrare l'intorno nella nuova soluzione trovata e nel ripetere le operazioni appena descritte. Un problema di questa metodologia riguarda gli ottimi locali, ovvero punti in cui la soluzione in considerazione, ad una data iterazione, ha un intorno dentro al quale non ne esistono di migliori. In queste situazioni l'algoritmo si ferma non riuscendo più a migliorare la soluzione. Un primo tentativo per risolvere tale problema consiste nell'allargare l'intorno di ricerca rendendolo *k-opt*, ma all'aumentare di k i tempi di calcolo aumentano notevolmente. E' quindi necessaria una tecnica tale da forzare uno spostamento nel caso in cui la soluzione attuale sia un ottimo locale. Per far ciò, si prevede di ricercare la soluzione migliore diversa da quella attuale, e così facendo se ne ottiene una peggiorativa ma che permette di muoversi evitando la situazione di stallo. Se riapplicassimo lo stesso metodo all'iterazione successiva però riotterremmo la soluzione precedente, ovvero l'ottimo locale da cui stiamo cercando di uscire. Per evitare questo loop si usano i Metaeuristici.

6.2.1 Tabu search

Questa tecnica [6] prevede di memorizzare, ad ogni mossa peggiorativa, la soluzione da cui siamo partiti, dichiarando la mossa che ci riporterebbe indietro *tabu*, ovvero vietata. Così facendo, nella ricerca di una nuova soluzione accettiamo eventuali mosse peggiorative mantenendo aggiornata una lista di divieti, detta *tabulist*. A furia di peggiorare, sperabilmente prima o poi troveremo una nuova soluzione, non vietata, che permette di migliorare il costo raggiungendo così un nuovo ottimo locale o magari un ottimo globale. A questo punto ripetiamo le operazioni precedenti tentando di uscire da questo nuovo punto di stallo ottenuto. Dopo alcune iterazioni la *tabulist* obbligherebbe l'ottimizzatore ad effettuare scelte forzate in quanto vietata ogni possibile decisione differente. Per evitare questo problema la *tabulist* viene considerata come oggetto dinamico in cui esiste una memoria e un parametro chiamato *tenure* indicante un determinato numero di iterazioni dopo il quale una soluzione vietata precedentemente torna ad essere accettabile. Questo parametro solitamente non viene mantenuto costante ma vien fatto variare alternando iterazioni con valori alti ad altre con valori bassi. Il vantaggio di questa oscillazione consiste nel non allontanare la soluzione da quella attuale nel caso in cui la tenure è alta,

fase detta di *intensificazione*, e di ampliare il campo di ricerca nel caso in cui sia bassa, fase di *diversificazione*.

6.2.2 Simulated annealing

Un altro metodo [7] per uscire da ottimi locali consiste in una tecnica molto simile ad una usata nella metallurgia per ottenere l'acciaio temperato. In questo settore si tenta di imitare i processi svolti dalla natura nel generare situazioni difficilmente riproducibili in maniera artificiale. Il meccanismo consiste nel cambiare la temperatura facendola oscillare in maniera regolare con l'obiettivo di raggiungere lo stato energetico minimo. Nella fase di innalzamento della temperatura si permette agli elettroni di muoversi e cambiare la loro posizione mentre nella fase di raffreddamento vengono fissati ottenendo un ottimo locale. La fase di riscaldamento è una buona tecnica usata dalla natura per diversificare. Per implementare questa metodologia dal punto di vista matematico introduciamo un parametro, la temperatura, che durante l'evoluzione dell'algoritmo viene ridotto. Nel nostro caso l'algoritmo funziona come uno di raffinamento che invece di cercare ad ogni iterazione la soluzione migliore di un intorno, valuta in maniera casuale un eventuale scambio che porta ad una variazione di costo. A questo punto accettiamo la mossa con una probabilità legata alla temperatura anche se la soluzione è peggiore della precedente. Se l'extra costo è eccessivo non effettuiamo lo scambio ma se è contenuto e la temperatura è alta lo scambio avviene. Se la soluzione è migliore della precedente lo scambio ha sempre successo e nel caso in cui il costo sia minore del più basso salvato si aggiorna l'ottimo globale mantenuto in memoria.

6.2.3 Variable Neighborhood Search

Questo metodo [8] prevede di uscire da eventuali ottimi locali attraverso una mossa *3-opt* fatta in maniera casuale. Così facendo, nell'iterazione successiva, la soluzione non potrà tornare in quella precedente in quanto troppo distante da raggiungere con una mossa *1-opt*. Questa tecnica non garantisce il non ripresentarsi dello stesso ottimo locale in quanto in più mosse potremmo ricapitare nella stessa situazione di stallo, tuttavia questa metodologia di diversificazione permette spesso, applicando una mossa al di fuori dell'intorno, di uscire da questa zona.

6.2.4 Algoritmi genetici

Gli algoritmi genetici nascono da una metafora biologica legata fortemente al concetto di evoluzione. In questo tipo di algoritmi, a differenza di quelli precedenti, non si lavora con una sola soluzione che viene man mano migliorata, ma con una popolazione di soluzioni diverse. Ipotizziamo di partire da una prima generazione composta da un numero di soluzioni pessime, nate in maniera casuale e quindi probabilmente nemmeno ammissibili. Le future generazioni verranno prodotte a partire dalle precedenti e, mediamente, le soluzioni saranno più vicine a quella ottima. Il termine che viene usato in questi casi è *fitness*, ed indica la misura di quanto la soluzione è buona e nel nostro caso indica il costo penalizzato che abbiamo introdotto (cambiamento di segno). Vogliamo creare un meccanismo che da una popolazione ad un'altra, ovvero da un'iterazione a quella successiva, migliori la *fitness* media. L'analisi di tale parametro quindi non consiste nello studio dei singoli individui ma dell'intera popolazione. Se si riesce a migliorare la *fitness* con buona probabilità, man mano che passano le generazioni ci avvicineremo ad una popolazione con individui molto interessanti. Ovviamente essendo interessati ad una singola soluzione ottima, ad ogni iterazione andremo a valutare la migliore soluzione nella popolazione corrente, e nel caso in cui abbia costo inferiore rispetto a quella in memoria, la salveremo. Per passare da una generazione alla successiva, presa una popolazione di individui, ne scegliamo due e li accoppiamo, producendo un figlio con delle caratteristiche ereditate dal patrimonio genetico dei due genitori. E' necessario quindi decidere con quale regola i genitori producono il figlio, e chiarire cosa si intende per patrimonio genetico di una soluzione. Supponiamo che il patrimonio genetico di un albero sia legato all'ordine con cui i vertici vengono visitati; per definire le caratteristiche che il figlio erediterà definiamo un punto di taglio casuale della lista e diciamo che egli prenderà una prima parte da un genitore e la restante dall'altro, questa operazione è detta di *crossover*. Ripetiamo la stessa operazione generando vari figli ampliando così la popolazione. Dopo aver fatto questa operazione per un determinato numero di volte, eliminiamo le peggiori soluzioni in modo tale da mantenere costante la taglia della popolazione tra una generazione e la successiva. Una tecnica simile però tenderebbe a far sparire nel tempo delle caratteristiche che al momento sembrerebbero pessime ma che in futuro potrebbero rivelarsi interessanti. E' conveniente quindi legare queste scelte a delle probabilità. Infine, potendosi presentare figli con delle caratteristiche erronee ereditate da entrambi i genitori, è necessario sistemare questi individui tramite una fase di restauro (*repair*) in modo tale da ripristinare

delle strutture ragionevoli all'interno della popolazione.

6.3 Test svolti

In questa sezione riportiamo i test svolti sugli algoritmi Tabu search e Variable Neighborhood Search da noi implementati. E' importante precisare che per ottimizzare al massimo le prestazioni di questi due algoritmi sono state effettuate delle parametrizzazioni che permettono di non verificare ad ogni cambio di soluzione il numero di crossing e di energia elettrica persa sull'intera soluzione in questione, ma di analizzare il grafo un numero di volte il più ridotto possibile. Più precisamente, una volta scelto un arco da rimuovere dalla soluzione attuale, viene simulata una sua cancellazione, si calcola il numero di crossing e la perdita di flusso senza la sua presenza. In seguito, una volta scelto l'arco da aggiungere, viene verificata la penalità generata da quest'ultimo inserimento, senza ricalcolare ad ogni tentativo il numero di tutti gli archi intersecanti e di energia persa nell'intera soluzione. Questa fase ha portato una singola mossa di scambio ad essere ottimizzata, riducendo il tempo di calcolo di circa 100 volte rispetto all'algoritmo implementato in partenza. Queste scelte di programmazione sono state attuate anche nell'implementazione del VNS rendendo i due algoritmi competitivi con le soluzioni fornite dall'utilizzo di CPLEX.
Osservando i dati raccolti durante le esecuzioni del programma si è potuto notare che la struttura regolare delle istanze prese in esame, dovuta alla disposizione uniforme delle turbine, causa un'ulteriore difficoltà da parte del metodo Tabu search nella fase di diversificazione. Essendo presenti numerose soluzioni con lo stesso costo si è deciso quindi di incentivare l'uscita dalla situazione di stallo, imponendo all'algoritmo la scelta di un arco che portasse ad un costo della soluzione differente rispetto all'iterazione precedente. Nonostante questa idea appariva inizialmente non migliorativa, in quanto la soluzione di costo uguale si ripresenta a iterazioni alterne (con archi differenti in quanto quelli precedenti restano vietati dalla *tabu list*), si è potuto concludere che questa tecnica di diversificazione porta soluzioni migliori sulla maggior parte delle istanze analizzate nei test svolti.
In Tabella 6.1 sono riportati i migliori costi ottenuti in 600 secondi con l'utilizzo dei metodi implementati. Con *Tabu search 1* si indica l'algoritmo classico, mentre con la tipologia 2 si indica la nostra variante. Per agevolarne un ulteriore confronto, gli stessi dati vengono riportati in un istogramma in Figura 6.3, limitando l'asse delle ordinate

a 10^8 in quanto soluzioni di costo superiore risultano inammissibili e quindi inutili da prendere in considerazione.

Istanza	**Tabu search 1**	**Tabu search 2**	**VNS**
Data 1	20110351.23	20121456.30	21201230.92
Data 2	21667861.74	21700013.90	22349898.02
Data 3	25122757.01	24834010.52	25393383.61
Data 4	27681068.47	27035458.21	26692581.42
Data 5	26458721.00	25682587.39	25717691.01
Data 6	25299852.70	25620592.06	26189993.49
Data 7	8573267.85	8564107.08	8564188.16
Data 8	8818966.46	8818966.46	8809664.96
Data 9	10125180.68	10080849.65	10066443.68
Data 10	10339153.37	10358255.74	10307495.40
Data 12	8621690.54	8631712.45	8604208.92
Data 13	8969179.44	8938011.67	8933494.42
Data 14	10671476.44	10185935.23	10173931.58
Data 15	10369414.92	10354903.00	10348430.54
Data 16	8063476.66	8054844.89	8054844.89
Data 17	8597903.94	8560008.61	8560008.61
Data 18	1009215201.35	1008670079.04	8369959.83
Data 19	1010159311.12	1009350669.11	9202178.95
Data 20	6047072531.11	2049753138.64	53720229.29
Data 21	6053390740.46	6053502221.20	53485389.42
Data 26	23108298.46	22876946.03	24301765.74
Data 27	24002878.34	24289724.54	24308867.15
Data 28	2031992599.37	1033674801.47	32778464.86
Data 29	4032563010.99	1035077902.62	35196878.47

Tabella 6.1: Risultati ottenuti in 600 secondi con Tabu search e VNS

Nel programma da noi implementato è stata resa possibile la graficazione dell'andamento del costo della soluzione, in funzione del tempo, nei due algoritmi. Come riportato in Figura 6.2, è facile notare come la mossa *3-opt*, utilizzata per diversificare nel VNS, rende la soluzione inammissibile portandola fuori dall'intorno dell'ottimo locale in cui si è bloccati. La continua necessità di diversificare, in istanze con disposizioni omogenee di turbine spesso equidistanti l'una dall'altra, rende però questo algoritmo non molto affidabile in quanto legato strettamente alla casualità della generazione degli archi della mossa *3-opt*. Nonostante questo, il VNS porta ad ottimi risultati in molte istanze.

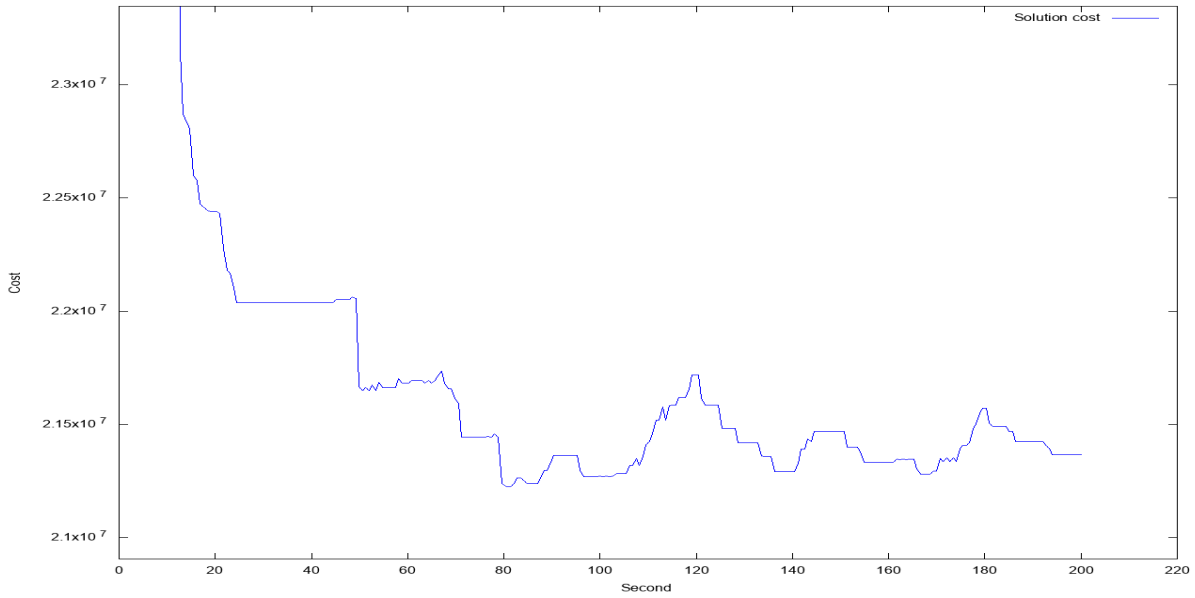

Figura 6.1: Andamento del costo in funzione del tempo con il Tabu search applicato su Data 1

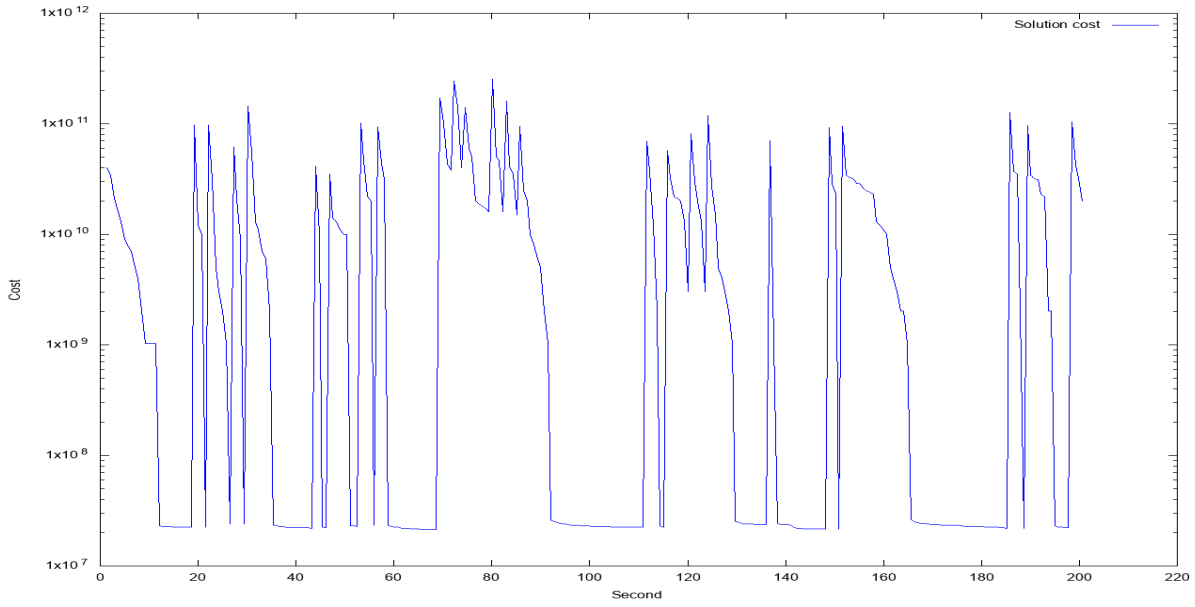

Figura 6.2: Andamento del costo in funzione del tempo con il VNS applicato su Data 1

Per le considerazioni fatte, nonostante la tecnica di diversificazione del VNS appaia a primo impatto migliore in base alle analisi svolte precedentemente sulla struttura regolare delle istanze analizzate, l'algoritmo Tabu search, con la tecnica di diversificazione citata

precedentemente, risulta essere più efficiente in termini di costo della soluzione rispetto al VNS nelle istanze con distribuzioni omogenee. La tecnica VNS risulta invece più efficiente sulle istanze con disposizioni meno regolari.

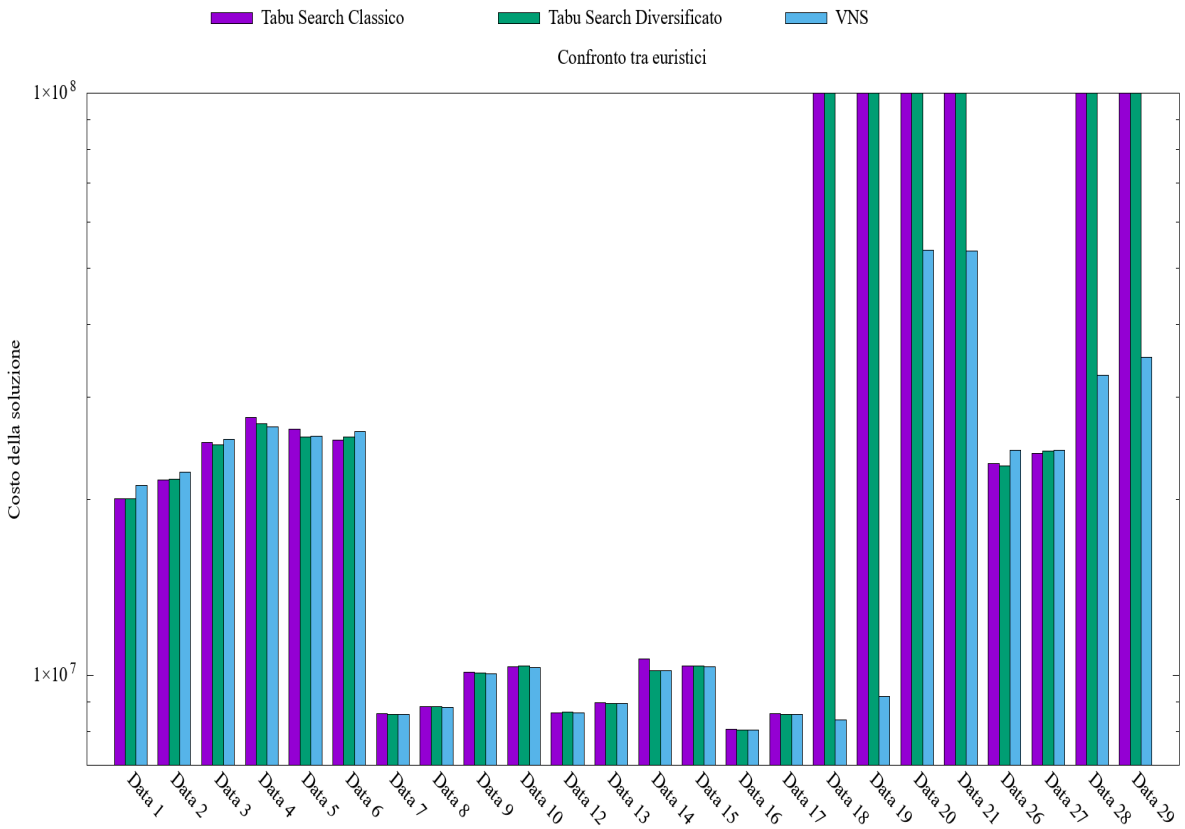

Figura 6.3: Risultati ottenuti in 600 secondi con Tabu search e VNS

7 Conclusioni

Dalle analisi svolte e riportate nei capitoli precedenti è possibile trarre varie conclusioni. Le istanze prese in considerazione per questo tipo di problema di ottimizzazione si sono distinte in varie classi di difficoltà legate al numero di turbine e al loro posizionamento nel parco eolico. Si è potuto notare come la distribuzione omogenea delle stesse abbia portato a tempi di risoluzione e di raggiungimento di una soluzione ottima in tempi elevati. Non è stato quindi identificato un metodo risolutivo migliore, in maniera generale, per ogni istanza. In particolar modo si è però potuto constatare che nell'utilizzo di CPLEX il supporto di RINS e di un rilassamento nel modello migliora notevolmente le prestazioni temporali della risoluzione del problema, in quanto si è notato che queste due tecniche agevolano la ricerca di una soluzione di riferimento da cui partire. Per quanto riguarda la fase di miglioramento di quest'ultima, invece, i vari metodi analizzati mostrano prestazioni differenti strettamente legate alla struttura dell'istanza analizzata e al tipo di rilassamento che si è deciso di utilizzare. Per la scelta di quest'ultima si sono rivelati ottimi i rilassamenti legati ai vincoli di eventuali perdite di flusso o di scelta forzata del cavo da utilizzare prediligendo il tipo più economico.
Per quanto riguarda gli euristici che non utilizzano di CPLEX, si è potuto verificare ancora più precisamente come le prestazioni dipendano dalla disposizione delle turbine, dal loro numero o, nel caso del Tabu search, dalla *tenure* utilizzata. La scelta di quest'ultimo parametro si è rivelata di importanza fondamentale, in quanto settaggi differenti hanno portato a risultati completamente discordi. Sono stati effettuati vari test per individua-

re un valore che permettesse di ottenere buone soluzioni nel maggior numero di istanze, ma i risultati ci hanno portato a concludere che una buona *tenure* è strettamente legata alle caratteristiche della singola istanza in considerazione e quindi difficilmente valida in generale. E' stato quindi scelto il valore migliore da noi considerato che portasse a buoni risultati in quasi tutti gli insiemi di dati analizzati. In Tabella 7.1 vengono riportati i risultati ottenuti dalle esecuzioni del Tabu search e del VNS da noi implementati, in confronto con quelli risultanti dall'utilizzo di CPLEX esposti nell'articolo [2] preso da noi come riferimento.

Istanza	**Articolo [2]**	**Tabu search 1**	**Tabu search 2**	**VNS**
Data 1	19436700.18	20035837.07	20018978.37	21201230.92
Data 2	21403410.11	21695549.10	21654712.88	22349898.02
Data 3	22611988.67	24649385.95	24819806.12	25393383.61
Data 4	24445688.02	26903696.51	26907729.06	26614883.12
Data 5	23482483.25	26465897.81	25514678.71	25431366.28
Data 6	24768927.72	26436740.51	25802581.63	25970420.53
Data 7	8555171.40	8573861.49	8571617.38	8555171.39
Data 8	8806838.99	8859155.01	8822759.73	8806839.00
Data 9	10056670.31	10092936.78	10078343.08	10056670.31
Data 10	10303320.51	10330486.48	10342929.95	10307495.40
Data 12	8604208.93	8614579.41	8607170.76	8604208.92
Data 13	8933494.59	8993837.29	8946647.97	8933494.42
Data 14	10173931.59	10195104.60	10198831.14	10173931.58
Data 15	10348430.63	10858470.00	10381158.94	10348430.54
Data 16	8054844.90	8054844.90	8054844.90	8054844.90
Data 17	8560008.68	8560015.53	8560008.61	8560008.61
Data 18	8357195.91	1008492903.80	8369959.83	8357195.90
Data 19	9178499.88	1009524745.89	1009449796.11	9178499.88
Data 20	38977593.84	6047064077.32	6053332764.17	51295133.82
Data 21	44857986.73	6053417966.86	6053375587.59	53485389.43
Data 26	22337935.84	22673780.60	22388691.79	23695163.83
Data 27	23362025.61	24001622.88	23654611.23	24308867.15
Data 28	26637602.25	32724083.75	33617718.29	32583209.26
Data 29	27295289.87	1033926470.29	32924347.80	33778636.01

Tabella 7.1: Confronto risultati ottenuti in 3600 secondi con Tabu search e VNS rispetto all'articolo [2]

Come è possibile osservare anche in Figura 7.1, i risultati riportati in [2] restano migliori rispetto a quelli ottenuti dalle nostre implementazioni, soprattutto nel caso di istanze difficili. Per quanto riguarda il confronto nel set di turbine comprese tra *Data 7* e *Data 18*, nelle quali CPLEX raggiunge e dimostra l'ottimo, i nostri euristici risultano molto buoni. Con queste analisi si è potuto quindi concludere che istanze con strutture molto regolari sono difficilmente risolvibili con gli algoritmi euristici da noi implementati. Per tali ragioni CPLEX risulta essere il miglior modo per risolvere questa tipologia di problema.

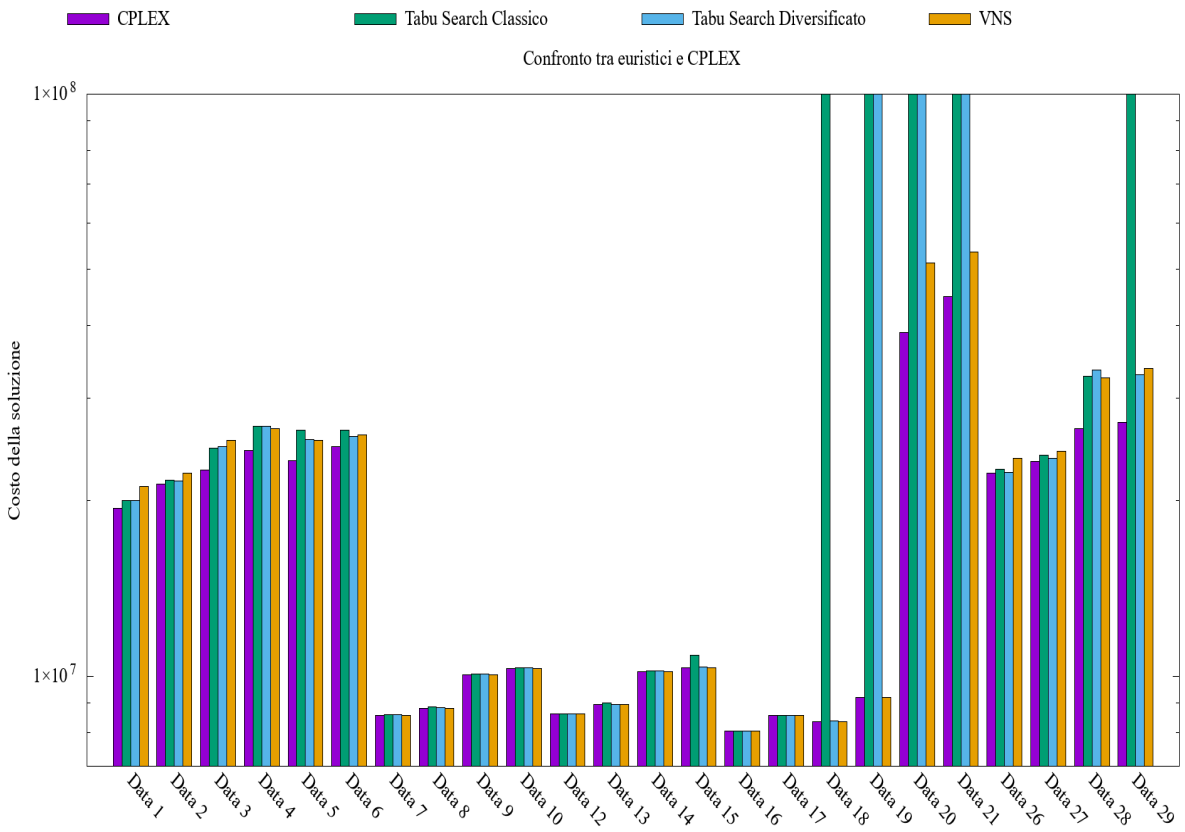

Figura 7.1: Risultati ottenuti in 3600 secondi con Tabu search e VNS rispetto a CPLEX

A

CPLEX

IBM ILOG CPLEX Optimization Studio, o più comunemente CPLEX, è un programma di ottimizzazione. Prende il nome dal metodo del simplesso (simplex method) implementato in linguaggio C, anche se al giorno d'oggi offre interfacce verso altri linguaggi come C++, C#, Phyton, Java e Matlab. Fu sviluppato da Robert E. Bixby e fu commercializzato a partire dal 1988 dalla CPLEX Optimization Inc., che venne successivamente acquistata, in un primo momento da ILOG nel 1997, e successivamente da IBM nel 2009. IBM ILOG CPLEX Optimizer risolve problemi di programmazione lineare, lineare intera e problemi di programmazione quadratica convessa. Esistono due modi differenti per interagire con il software. Il primo avviene tramite la scrittura di un modello matematico in un file di testo in formato LP che verrà letto e risolto da CPLEX in interattivo. Il secondo, invece, prevede l'utilizzo delle API del risolutore per l'implementazione di un interfaccia attraverso codice sorgente. La risoluzione utilizza le varianti primale o duale del metodo del simplesso. CPLEX viene utilizzato in vari ambiti rendendo problemi aziendali veri e propri problemi matematici modellizzati e ottimizzati. Questo meccanismo permette alle più grandi aziende al mondo di risparmiare milioni di euro.

B
Gnuplot

Gnuplot è un programma che permette la rappresentazione di funzioni matematiche in due o tre dimensioni. E' disponibile per diversi sistemi operativi e possiede un'interfaccia a riga di comando. Il suo utilizzo permette di esportare grafici in differenti formati, nel nostro caso utilizzeremo il PNG. Una volta scaricato e installato il programma tramite il sito `http://www.gnuplot.info` è possibile da subito utilizzare il software aprendo il terminale e digitando il comando gnuplot. Una prima rappresentazione basilare può essere ottenuta tramite il comando *plot sin(x)* ottenendo il risultato riportato in Figura B.1.

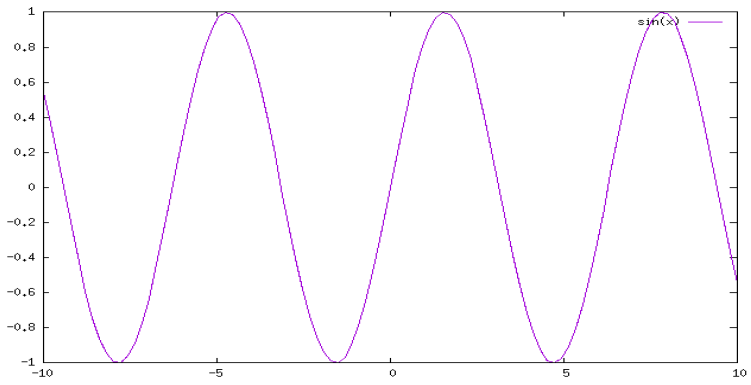

Figura B.1: Rappresentazione sin(x)

Attraverso l'help o tramite diverse guide o documentazioni [11] disponibili online è possibile personalizzare il grafico che verrà stampato tramite la funzione *plot*.

Per esempio, tramite delle impostazioni di gnuplot è possibile decidere se rappresentare il grafico tramite una linea di un determinato spessore, tipo e colore. Questo viene fatto usando un comando del tipo *plot sin(x) with linespoints linecolor rgb 'blue' linewidth 2 pointtype 7 pointsize 1* ottenendo il seguente risultato

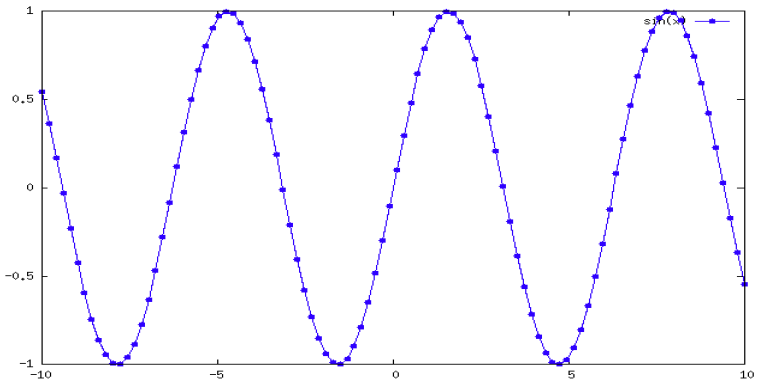

Figura B.2: Rappresentazione personalizzata sin(x)

Lo stesso risultato viene raggiunto con il comando più sintetico *plot sin(x) with linespoints lc rgb 'blue' lw 2 pt 7 ps 1*. Se richiesto dall'utente, le impostazioni possono essere settate all'apertura del programma rendendole valide per tutte le rappresentazioni che verranno effettuate in quella sessione. Un'esempio di istruzione utile riguarda la scelta del terminale da voler utilizzare, impostando così il formato di output del plot.

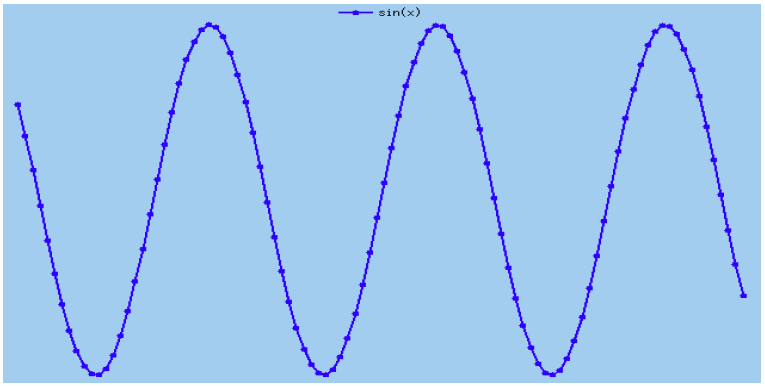

Figura B.3: Rappresentazione personalizzata sin(x)

Nel caso in cui si voglia per esempio impostare come formato di uscita il PNG basta inserire l'istruzione *set terminal png*, aggiungendoci *enhanced background rgb '#ABCDEF'* è possibile inoltre colorare lo sfondo della rappresentazione con una tinta scelta.

Il codice all'interno degli apici rappresenta il colore scelto in formato rgb. I comandi *unset border*, *unset xtics*, *unset ytics* e *set key reverse above Left width 1* permettono di eliminare il bordo della figura, gli assi cartesiani e di aggiungere una zona sopra l'immagine dove raccogliere le leggende delle linee rappresentate. La Figura B.3 è il risultato derivante dall'aggiunta di questi ultimi settaggi.
Nel caso specifico di nostro interesse abbiamo deciso, per comodità, di creare un file in formato *.gp* che, fornito in input a gnuplot, permette al software di seguire in ordine le istruzioni scritte. Così facendo l'unica istruzione necessaria per la rappresentazione di uno o più grafici è *gnuplot nome_file.gp*. Questo metodo permette di creare veri e propri programmi script con gnuplot. La nostra scelta è stata quella di plottare le turbine della wind farm trattata con dei puntini, i cavi scelti nel collegamento di un colore differente in base alla capacità di supporto e infine con dei punti neri più grossi eventuali crossing. Il risultato ottenuto è il seguente

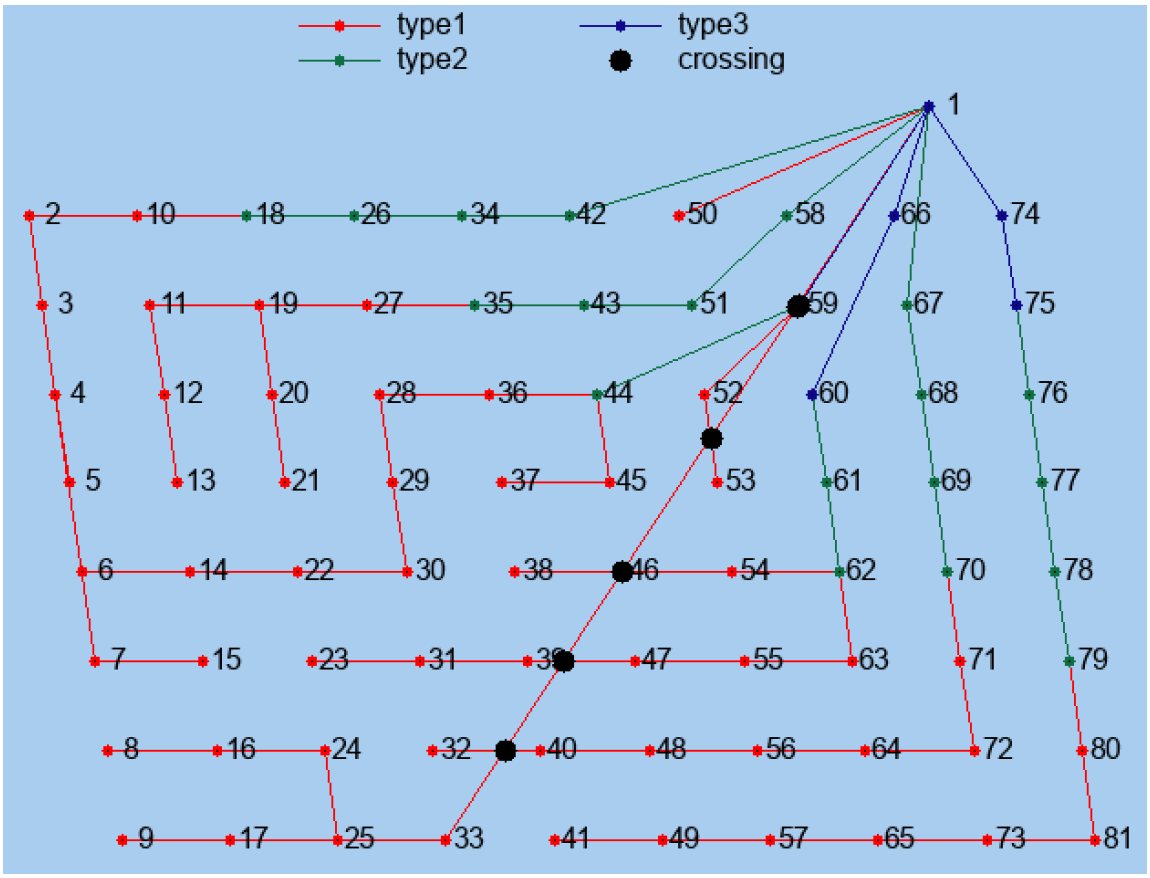

Figura B.4: Rappresentazione wind farm

Vengono riporte le istruzioni scritte nel *printplot.gp* per ottenere il precedente risultato.

```
set terminal png enhanced background rgb '#ABCDEF'
set key reverse above Left width 1
set style line 1 lc rgb 'red' lt 1 lw 1 pt 7 ps 1
set style line 2 lc rgb '#177245' lt 1 lw 1 pt 7 ps 1
set style line 3 lc rgb '#120A8F' lt 1 lw 1 pt 7 ps 1
set style line 4 lc rgb '#FF9933' lt 1 lw 1 pt 7 ps 1
set style line 5 lc rgb '#FF2400' lt 1 lw 1 pt 7 ps 1
set style line 6 lc rgb '#6F00FF' lt 1 lw 1 pt 7 ps 1
set style line 7 lc rgb '#2F2F2F' lt 1 lw 1 pt 7 ps 1
set style line 8 lc rgb '#FFD800' lt 1 lw 1 pt 7 ps 1
set style line 9 lc rgb '#DF73FF' lt 1 lw 1 pt 7 ps 1
set style line 10 lc rgb '#7FFF00' lt 1 lw 1 pt 7 ps 1
set style line 11 lc rgb '#F0DC82' lt 1 lw 1 pt 7 ps 1
set style line 12 lc rgb '#3D2B1F' lt 1 lw 1 pt 7 ps 1
set style line 13 lc rgb '#908435' lt 1 lw 1 pt 7 ps 1
set style line 14 lc rgb '#9966CC' lt 1 lw 1 pt 7 ps 1
set style line 15 lc rgb '#E52B50' lt 1 lw 1 pt 7 ps 1
unset border
unset xtics
unset ytics
set output "wind_farm.png"
plot for [k=1:3] "plot.txt" index "type".k with linespoints ls k title "type".k, "plotcrossing.txt"
with points pt 7 ps 2 lc rgb 'black' title "crossing", "../../Data/data_01.turb" with points pt 7 ps 0
 lc rgb 'black' notitle, "../../Data/data_01.turb" using 1:2:($0+1) with labels offset 1.5 notitle
exit
```

I file *plot.txt* e *plotcrossing.txt* riportano le coordinate delle turbine collegate tra loro e dei crossing. Il primo file è formattato in modo da avere nella prima colonna la coordinata X e nella seconda la Y di un singolo punto; due coppie di punti in righe successive non separate da una linea bianca rappresentano due turbine che dovranno essere collegate tra loro. La linea da utilizzare nella rappresentazione del cavo viene identificata dalla sezione di file in cui la coppia di punti viene trovata.

```
# type1
423974 6151448
424534 6151447

424042 6150891
426842 6150891


# type2
424315 6148668
428031 6150335

425859 6149780
428636 6152126


# type3
427471 6150335
427962 6150890

427894 6151447
428636 6152126
```

Nel secondo file è necessario indicare nella prima colonna la coordinata X e nella seconda la Y che identificano il punto in cui avviene il crossing.

```
425562.406065  6150891.000000
424671.040844  6150334.666642
425231.081688  6150334.333285
424471.205470  6149224.000000
425554.409718  6149224.000000
424608.746600  6148799.774915
425554.409718  6149224.000000
425907.621118  6149382.450862
```

Una seconda versione della rappresentazione della wind farm da noi proposta, presenta due varianti rispetto alla B.4. Abbiamo voluto migliorare il plot in modo da utilizzare delle frecce indicanti il verso del flusso di corrente e sono state aggiunte delle immagini indicanti il tipo di punto rappresentato, risulta molto più intuibile ora capire come le turbine vengano collegate tra loro conducendo l'energia elettrica verso la substation. Per far ciò è stata cambiata la struttura del file *plot.txt* e del *printplot.gp*. Si riportano le istruzioni utilizzate e il formato dei dati di input di plot.txt

```
set terminal png size 1280, 960 enhanced background rgb '#ABCDEF'
set key reverse above Left width 1
set style line 1 lc rgb 'red' lt 1 lw 1 pt 7 ps 1
set style line 2 lc rgb '#177245' lt 1 lw 1 pt 7 ps 1
set style line 3 lc rgb '#120A8F' lt 1 lw 1 pt 7 ps 1
set style line 4 lc rgb '#FF9933' lt 1 lw 1 pt 7 ps 1
set style line 5 lc rgb '#FF2400' lt 1 lw 1 pt 7 ps 1
set style line 6 lc rgb '#6F00FF' lt 1 lw 1 pt 7 ps 1
set style line 7 lc rgb '#2F2F2F' lt 1 lw 1 pt 7 ps 1
set style line 8 lc rgb '#FFD800' lt 1 lw 1 pt 7 ps 1
set style line 9 lc rgb '#DF73FF' lt 1 lw 1 pt 7 ps 1
set style line 10 lc rgb '#7FFF00' lt 1 lw 1 pt 7 ps 1
set style line 11 lc rgb '#F0DC82' lt 1 lw 1 pt 7 ps 1
set style line 12 lc rgb '#3D2B1F' lt 1 lw 1 pt 7 ps 1
set style line 13 lc rgb '#908435' lt 1 lw 1 pt 7 ps 1
set style line 14 lc rgb '#9966CC' lt 1 lw 1 pt 7 ps 1
set style line 15 lc rgb '#E52B50' lt 1 lw 1 pt 7 ps 1
unset border
unset xtics
unset ytics
set output "wind_farm.png"
plot "../substation.png" binary filetype=png origin=(428636.000000, 6152176.000000) with rgbimage
notitle,
"../turbina.png" binary filetype=png origin=(423874.000000, 6151448.000000) with rgbimage notitle,
"../turbina.png" binary filetype=png origin=(423942.000000, 6150891.000000) with rgbimage notitle,
"../turbina.png" binary filetype=png origin=(424011.000000, 6150335.000000) with rgbimage notitle,
"../turbina.png" binary filetype=png origin=(424079.000000, 6149780.000000) with rgbimage notitle,
"../turbina.png" binary filetype=png origin=(424147.000000, 6149224.000000) with rgbimage notitle,
                                        ...
                                        ...
                                        ...
                                        ...
                                        ...
"../turbina.png" binary filetype=png origin=(429392.000000, 6147541.000000) with rgbimage notitle,
for [k=1:3] "plot.txt" index "type".k using 1:2:3:4 with vectors head filled ls k title "type".k,
"../../Data/data_01.turb" using 1:2:($0+1) with labels offset 1.5 notitle,
"plotcrossing.txt" with points pt 7 ps 2 lc rgb 'black' title "crossing"
exit
```

```
# type1
423974 6151448 560 -1
424042 6150891 -68 557
424111 6150335 -69 556
424179 6149780 -68 555

# type2
426214 6151447 560 0
426774 6151447 560 0
426842 6150891 560 0
426911 6150335 560 0

# type3
427471 6150335 491 555
427962 6150890 674 1236
428454 6151447 182 679
429014 6151447 -378 679
```

Il risultato ottenuto è il seguente

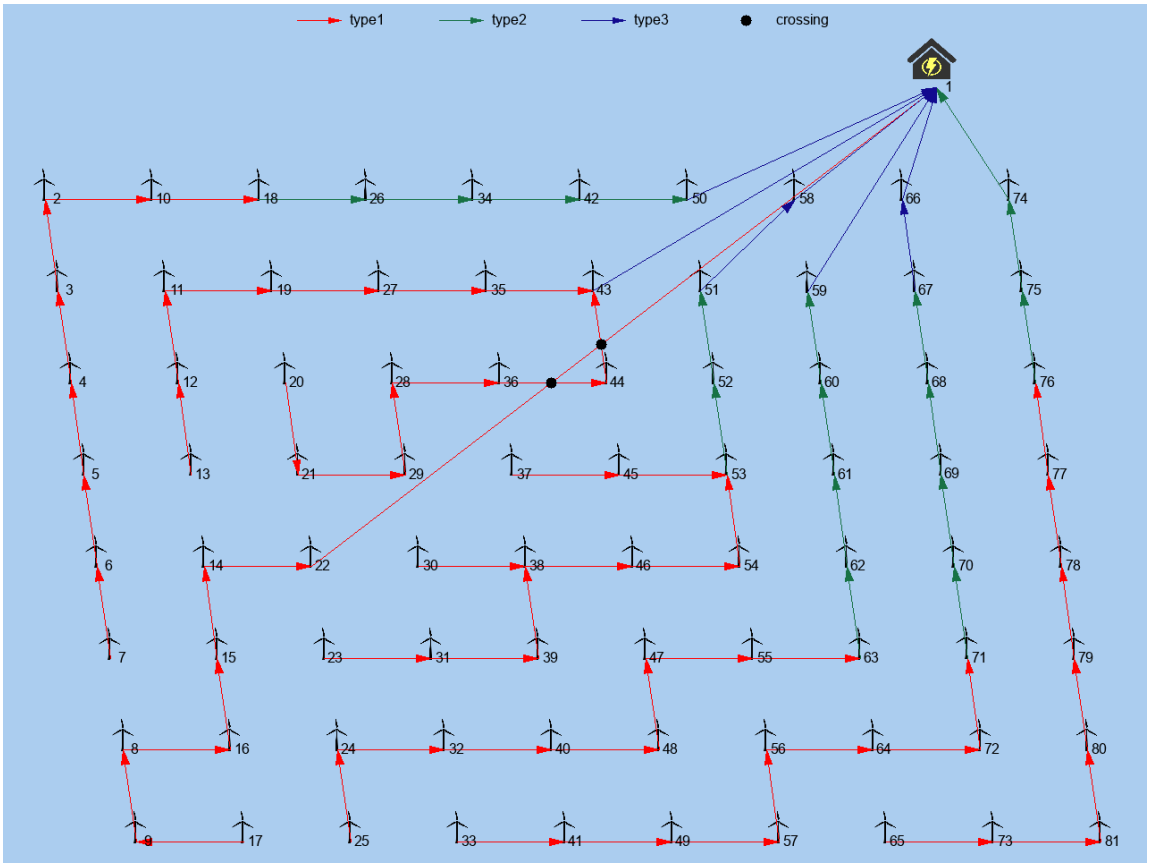

Figura B.5: Rappresentazione wind farm

Per ulteriori informazioni sull'utilizzo di gnuplot o per la creazione dei file *printplot.gp*, *plot.txt* e *plotcrossing.txt* si rimanda al codice del programma sviluppato.

Riferimenti

[1] CPLEX Optimizer. https://www.ibm.com/analytics/data-science/prescriptive-analytics/cplex-optimizer

[2] Fischetti M., Pisinger D., *"Optimizing wind farm cable routing considering power losses"*, European Journal of Operational Research, 2017.

[3] Danna E., Rothberg E., Le Pape C., *"Exploring relaxation induced neighborhoods to improve MIP solutions"*, Mathematical Programming pag:71-90, 2005.

[4] Rothberg E., *"An Evolutionary Algorithm for Polishing Mixed Integer Programming Solutions"*, INFORMS Journal on Computing pag:534-541, 2007

[5] Fischetti M., Lodi A., *"Local branching"*, Springer-Verlag Berlin Heidelberg, 2003.

[6] Glover F., Laguna M., *"Tabu search"*, Kluwer Academic Publishers Dordrecht, 1997.

[7] Kirkpatrick S., Gelatt C.D., Vecchi M.P., *"Optimization by simulated annealing"*, Science 220 pag:671-680, 1983.

[8] Hansen P., Mladenovi N., Uroevi D., *"Variable neighborhood search and local branching"*, Computers & Operations Research vol:33 pag:3034-3045, 2006.

[9] Cluster di calcolo dipartimentale Blade. https://www.dei.unipd.it/bladecluster

[10] Gnuplot. http://gnuplot.sourceforge.net

[11] Documentazione Gnuplot. http://www.gnuplot.info/docs_5.2/Gnuplot_5.2.pdf