POLITECNICO DI TORINO

Collegio di Ingegneria Informatica

Tesi di laurea magistrale

Utilizzo dell'algoritmo Ant Colony per la minimizzazione delle perdite in un impianto di distribuzione di energia elettrica

Relatore

prof. Giovanni Malnati

Candidato

Marco Di Blasi

Dicembre 2017

Indice

Elenco delle figure

Introduzione

Il presente lavoro di tesi ha avuto come obiettivo la formulazione di un nuovo metodo per il risparmio energetico, nell'ambito della distribuzione dell'energia elettrica, per mezzo di un algoritmo metaeuristico: l'**Ant Colony Optimization**.
Le cause di dissipazione di energia elettrica sono molteplici. Questo lavoro si occupa delle perdite, nei sistemi in corrente alternata, causate dalle caratteristiche dei conduttori e dalla presenza di Potenza Reattiva, che produce un flusso di corrente supplementare, non utilizzata dai carichi ed incrementa la dissipazione per effetto Joule. L'idea nasce dalla possibilità di poter gestire la generazione di potenza elettrica in funzione della richiesta di potenza assorbita dai carichi in una rete di distribuzione radiale di dimensioni limitate, esplorando, tramite l'algoritmo Ant Colony, i migliori profili di "generazione" dell'energia con l'obiettivo di minimizzare le perdite.
Per creare una simulazione di quello che potrebbe essere un sistema di gestione della distribuzione di energia, sono stati sviluppati: un programma in C# *.net* che rappresenta un server; alcuni servizi esterni (REST service, Dashboard Web) che rappresentano i client che comunicano con il server. Il lavoro di tesi qui presentato sarà suddiviso in una prima parte, nella quale verrà illustrato il concetto di "Swarm Intelligence" (Capitolo 1), ponendo l'attenzione su una delle sue manifestazioni principali quale l'algoritmo Ant Colony; una seconda sezione dedicata alla distribuzione di energia elettrica nelle reti radiali (Capitolo 2) e all'algoritmo Backward Forward Sweep, utilizzato in questo lavoro per il calcolo delle perdite; una terza parte illustrerà l'implementazione di un programma che combina i 2 algoritmi sopracitati e mostra la variazione del flusso di energia tramite un dashboard (Capitolo 3).

CAPITOLO 1

La Swarm Intelligence e l'Ant Colony

Swarm intelligence (intelligenza dello sciame) è un espressione che indica letteralmente la capacità di uno sciame di intraprendere azioni complesse sulla base di un intelligenza collettiva.

Prende origine dai modelli di comportamento degli insetti sociali e può essere definita come un intelligenza collettiva emergente in gruppi di agenti semplici. I modelli più famosi sono quelli che si ispirano alla ricerca del cibo delle formiche o alla costruzione dei nidi di termiti ed ancora la disposizione di aree specifiche negli alveari da parte delle api. Ognuno di questi modelli condivide alcune proprietà quali flessibilità, agenti semplici con limitate capacità di ragionamento individuale, interazioni dirette e indirette fra gli agenti, suddivisione e distribuzione del lavoro. Alcuni esempi di problemi risolti dall'intelligenza collettiva degli insetti sociali possono essere il raggruppamento di oggetti, la ricerca di cibo, il trasporto cooperativo come esempio di suddivisione del lavoro.

L'auto-organizzazione è uno dei principi su cui si basa la risoluzione di un problema complesso per mezzo dell'intelligenza collettiva; questo rappresenta un insieme di azioni e meccanismi globali attuati di conseguenza alle interazioni, dirette e indirette, dei componenti locali. Le interazioni fra agenti possono dunque essere di diverso tipo; innanzi tutto possono essere interazioni semplici fra agenti o interazioni multiple fra sistemi a loro volta composti da gruppi di agenti; possono essere dirette, indicando un'azione esatta da far intraprendere ad un agente, o indirette, come ad esempio la decisione di un agente di muoversi in un ambiente sulla base di un intorno modificato da se stesso o da un altro agente in un istante di tempo precedente. Quest'ultimo concetto è detto

Stigmergia ed è un vero e proprio metodo di comunicazione basato sull'alterazione di un ambiente condiviso.
Un altro elemento fondamentale dell'intelligenza collettiva è la capacità individuale di discriminazione; essa rappresenta l'abilità di un insetto sociale di discriminare fra "compagni di tana" (nestmtes) e "non compagni di tana" (non-nestmates). In una società di formiche, individui provenienti da gruppi differenti riconoscono la propria identità chimica fra le altre; tuttavia i gruppi possono cooperare e intraprendere azioni collettive; se, ad esempio, consideriamo uno scenario in cui più gruppi di formiche, provenienti da più tane differenti, trovano cibo in una sorgente comune, ognuna delle formiche singolarmente saprà verso quale tana portare il cibo raccolto.
In questo capitolo, molti degli argomenti sono stati ripresi dal libro di Marco Dorigo e Thomas Stützle [1].

1.1 Ant Colony Optimization

L'Ant Colony Optimization (ACO) è uno fra gli algoritmi più noti nell'insieme degli Ant Algorithms e presenta molte varianti. Questi algoritmi si ispirano al comportamento delle formiche in fase di foraggiamento immaginando ogni formica come un singolo agente di un sistema distribuito.
Ogni formica comunica con le altre e con l'ambiente per mezzo di una traccia chimica rilasciata sul suo percorso. La sostanza chimica che genera la traccia è il *feromone*. Questa modalità di comunicazione è molto lontana da quella a cui siamo abituati, basata prevalentemente su senzi visivi e auditivi. Alcune spcie di formiche come le Lasius niger usano dunque il feromone per tracciare un percorso, ad esempio, dalla tana alla sorgente di cibo. In questo modo le altre formiche possono avvertire la presenza di feromone e andare a raccogliere cibo da una sorgente scoperta da un'altra formica. Le formiche che seguono una traccia di feromone, rilasciano a loro volta una traccia sul terreno. In questo modo il feromone presente sul terreno viene incrementato nel tempo e permette agli sciami di convergere verso una sorgente di cibo in numero sempre maggiore.
L'osservazione del comportamento delle formiche ha portato alla formulazione di molti esperimenti riguardanti la scelta del percorso. Uno degli esperimenti più famosi è il **Double Bridge Experiment** in cui si è osservata la reazione delle formiche davanti ad alcune situazioni di scelta, nella quale i percorsi a disposizione avevano lunghezze

diverse. In figura un esempio di percorso a Double Bridge con rami equivalenti (a) e rami di lunghezze diverse (b).

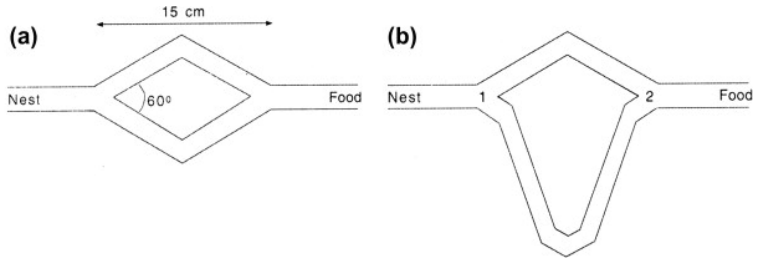

Figura 1.1: Double Bridge

Nel primo esperimento venivano proposti due percorsi *tana-cibo* della stessa lunghezza e si è osservato che, dopo una fase iniziale di scelta randomica, le formiche convergevao in maggior numero verso uno dei due percorsi. Ripetendo questo esperimento si è notato che la scelta di una delle due strade, da parte delle formiche, era puramente randomica. Nel secondo esperimento i percorsi proposti alle formiche erano uno il doppio dell'altro. Anche in questo caso, durante la fase iniziale, i percorsi venivano scelti casualmente, ma la convergenza finale di tutto lo sciame si dirigeva verso il percorso più breve. Questo perchè nonostante la quantità di feromone fosse rilasciata in parti uguali su entrambi i percorsi, il tempo di percorrenza sul cammino più breve era dimezzato rispetto al cammino più lungo. Per questo motivo sul percorso più breve l'incremento di feromone era molto più veloce e permetteva alle formiche di scegliere il cammino più breve con una probabilità sempre più alta. Un terzo esperimento ha evidenziato un limite di questa metodologia di comunicazione dello sciame: in questo caso, all'inizio dell'esperimento vi era un solo percorso disponibile e solo dopo 30 minuti veniva aggiunto un percorso più breve. Si è osservato che solo sporadicamente le formiche sceglievano il nuovo cammino aggiunto; questo perchè nei primi 30 minuti il rilascio di feromone sul primo percorso era stato massivo, essendo l'unico disponibile.
Il double bridge experimente ha evidenziato come in molti casi, seguendo la logica delle formiche, si possa trovare un cammino minimo sulla base di scelte determinate da informazioni locali. Riportando questa logica all'interno di un grafo, è possibile creare un modello di formica artificiale che possa scovare un percorso minimo fra due nodi. Risulta però necessario inserire un po' più di intelligenza in modo da evitare di incontrarsi con il limite descritto dal terzo esperimento.

1.1.1 Logica dell'Ant Colony in un grafo

All'interno di un grafo in cui le formiche, partendo da un determinato nodo, sono libere di scegliere localmente su quale arco procedere è necessario inserire una discretizzazione del tempo. Questo ci permette di controllare ad ogni istante t la scelta di ogni formica. La decisione delle formiche è probabilistica e la probabilità è calcolata in funzione della quantità di feromone depositato. Su ognuno dei nodi i del grafo, esisteranno quindi una probabilità $p_{is}(t)$ di scegliere il cammino più corto e una probabilità $p_{il}(t)$ di scegliere il cammino più lungo, ad ogni istante t.
Facendo rifirimento al Double Bridge Experiment, uno dei modi in cui si può modellare la configurazione con 2 rami di lunghezze diverse è quello di spezzare il ramo più lungo in una sequenza di più rami. Questa modellazione permetterà di allungare il tempo di *update* del feromone su quello che era il cammino più lungo, in quanto per aggiornare la quantità di feromone su tutti i rami che compongono il cammino lungo, serviranno tanti istanti di tempo quanti sono i nodi inseriti per spezzettare il ramo iniziale.
Questa logica funziona con l'obiettivo di trovare un percorso minimo fra tanti, solo se ogni percorso è indipendente da ogni altro. Nel caso in cui il grafo proposto sia conesso, spezzando quindi il vincolo di inidipendenza, sorge il problema dei cicli. Una formica artificiale potrebbe ciclare su una serie di archi del grafo allungando di molto il percorso verso il nodo finale, in alcuni casi senza mai raggingerlo. In figura un esempio di grafo con cammini non indipendenti.

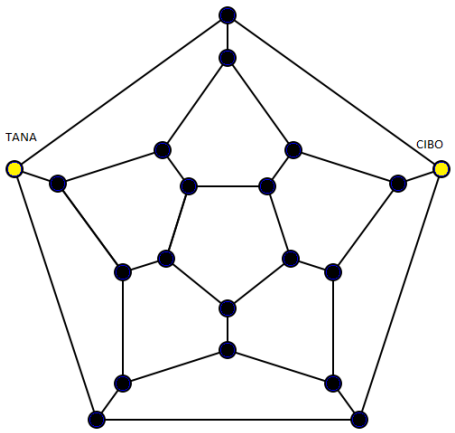

Figura 1.2: Grafo Connesso

Per ovviare a questo problema ogni formica artificiale viene munita di una semplice memoria utilizzata per ricordare il cammino parziale percorso fino a quell'istante di tempo.
L'aggiunta di una semplice capacità di memorizzazione permette alle formiche di non convergere in un *loop*, impostando a probabilità di scelta nulla gli archi che conducono ad un nodo precedentemente visitato. Anche la modalità di aggiornamento del feromone depositato subisce un cambiamento. Utilizzando la memoria è possibile aggiornare il feromone durante il cammino di ritorno, ripercorrendo tutti i nodi in memoria e calcolandone la quantità da depositare basandosi ad esempio sulla bontà del percorso. Se si vuole ad esempio trovare un percorso minimo in un grafo in cui ad ogni arco è associata una lunghezza, la quantità di feromone da depositare può essere calcolata sulla base del totale delle lunghezze percorse durante il cammino di andata.
Facendo riferimento alla terza parte del Double Bridge Experiment è necessiaro un ultimo accorgimento sull'aggiornamento della quantità di feromone depositato su un cammino, per evitarne la crescita esponenziale nel tempo. Per evitare che alcuni cammini nel grafo rimangano inesplorati, bisogna fare in modo che la quantità di feromone non cresca esponenzialmente sugli archi precedentemente scelti per mezzo di reiterati passaggi di nuove formiche. In pratica bisogna assicurarsi che il calcolo della probabilità di scelta di ogni arco non dia mai come risultato un numero troppo vicino allo zero o troppo vicino all'1. Sono stati formulati due metodi che aiutano il sistema ad ovviare a questo problema. Il primo consiste nell'inizializzare la quantità di feromone sugli archi ad un numero diverso da zero (la scelta di questo numero fa parte dei parametri metaeuristici che dipendono da numero di formiche utilizzate, numero di nodi da attraverare, e altri fattori). Il secondo metodo consiste nell'
evaporazione del feromone; si può decidere infatti che ad ogni aggiornamento del feromone su un cammino venga associata, oltre alla quantità di nuovo feromone da depositare in base alla bontà di un percorso, una quantità di feromone che evapora e che matematicamene viene sottratta a quella presente. Il valore di questa quantità è un altro dei parametri metaeuristici che dipendono dalla tipologia di problema affrontato e che determinano la velocità di convergenza dell'algoritmo.

1.2 Parametri dell'ACO

Come menzionato nella sezione precedente, essendo un algoritmo metaeuristico, il corretto funzionamento dell'ACO dipende da molti parametri. Di seguito saranno elencati i parametri con una breve spiegazione sul loro utilizzo.

- **Numero di formiche N**

  Questo prametro decide quante formiche partono dalla tana e cercano un nuovo percorso ad ogni iterazione; N deve essere dimensionato sulla base della grandezza del grafo da esplorare.

- **Numero di iterazioni**

  Questo parametro rappresenta il criterio di *stop* dell'algoritmo; se non viene utilizzato, l'algoritmo non ha modo di decidere quando fermarsi. Siccome è conveniente non far arrivare a zero la quantità di feromone presente su ogni cammino per mezzo dell'evaporazione e siccome le formiche artificiali non hanno modo di capire quando il "cibo fittizio" sia finito nella sorgente, è necessario introdurre questo parametro per terminare la ricerca.

- **Coefficiente di incremento del feromone Q**

  Questo coefficiente rappresenta il fattore di incremento del feromone sui cammini e determina la quantità da depositare su un preciso percroso.

- **Coefficiente di decremento del feromone ρ**

  Al contrario di Q, questo coeficiente determina la quantità di feromone che evapora ad ogni iterazione. E' un numero vincolato a rimanere fra 0 e 1 in quanto la quantità di decremento è determinata dalla formula (in questo caso semplificata):

  incremento = (1 - ρ) * feromone presente

- **Coefficente di influenza del feromone α**

  Ogni volta che una formica si trova davanti ad un bivio, viene calcolata la probabilità di scegliere una strada rispetto ad un'altra sulla base del feromone depositato. Questo coefficiente l'influenza della quantità di feromone sul calcolo della probabilità.

- **Coefficente di influenza della bontà del cammino** β

  Anche questo parametro è utilizzato durante il calcolo della probabilità di scelta di un cammino ma nel caso specifico di questa tesi non è influente in quanto dipendente dalla lunghezza dell'arco su cui viene applicata la formula. Il grafo che sarà presentato in seguito e che rappresenta tutti i cammini delle formiche per la risoluzione del problema analizzato in questo lavoro, contiene solo archi con lunghezza unitaria.

In seguito si vedrà come viene utilizzato matematicamente ognuno di questi parametri.

1.3 ACO su TSP e varianti

In questa tesi il programma sviluppato si basa su un algoritmo esistente applicato al problema del Traveling Salesman Problem (TSP) [2]. Di seguito sarà presentato questo algoritmo insieme ad alcune sue caratteristiche e varianti.
L'algoritmo ACO venne utilizzato per la prima volta nella ricerca della soluzione del TSP (il il problema in questione rappresenta la ricerca del cammino a costo minore in un grafo non orientato completamente connesso, partendo da un determinato nodo e visitando tutti gli altri nodi al più una volta). Questa prima formulazione fu chiamata **Ant System** e trovò molte applicazioni per la risoluzione di molti problemi esistenti e classificabili come sottoproblemi del TSP. La caratteristica principale di questo algoritmo risiede nella capacità di trovare una soluzione ottima del TSP con un tempo di convergenza molto più breve rispetto ad altri. Il problema analizzato in questa tesi appartiene all'insieme dei sottoproblemi del TSP, questo ci garantisce di trovarne una soluzione ottima. In tutti gli algoritmi ACO sviluppati per la soluzione del TSP la traccia di feromone è associata agli archi, se quindi definiamo τ_{ij} come la quantità di feromone depositata sull'arco che unisce i nodi i e j, essa rappresenterà anche la desiderabilità di una formica di scegliere quell'arco.
La costruzione di un cammino dal nodo di partenza al nodo di arrivo (quest'ultimo è rappresentato dall'ultimo nodo visitato) segue una procedura per ogni formica che può essere sintetizzata come segue:

- Scelta del nodo di partenza delle formiche base ad un determinato criterio;

- Utilizzo del feromone τ_{ij} presente sugli archi per calcolare la probabilità di scelta su ogni arco collegato al nodo in cui la formica si trova e memorizzazione di tutti i nodi precedentemente visitati;
- Dopo aver visitati tutti i nodi appartenenti al grafo, la formica ripercorre all'indietro il cammino memorizzato depositando il feromone.
- Ogni volta che tutte le formiche sono tornate indietro viene applicato un aggiornamento globale del feromone che rappresenta l'evaporazione.

Inizialmente esistevano tre diverse versioni dell'Ant System: *ant-density, ant-quantity* e *ant-cycle*. Nelle prime due, il feromone viene depositato sul cammino ad ogni scelta di arco. Nell'ant-cycle invece il feromone viene depositato solo sul cammino di ritorno. Le prime due varianti furono abbandonate in quanto presentavano minori performance; per questo motivo si seguirà la terza variante.
Le fasi più importanti dell'Ant System sono la costruzione del percorso da nodo iniziale a nodo finale e l'aggiornamento del feromone. Non è da sottovalutare però l'inizializzazione del feromone su tutto il grafo; questo valore deve essere dimensionato in base alla quantità stimata di deposito di feromone di ogni formica ad ogni iterazione. Questa stima può essere calcolata per mezzo della formula:

$$\tau_0 = \frac{m}{C} \tag{1.1}$$

dove m è il numero di formiche e C è la lunghezza di un percorso completo.

Costruzione del percorso

Nell'Ant System ogni formica costruisce il suo percorso in concorrenza alle altre. La scelta dell'arco da percorrere dipende dalla probabilità associata all'arco stesso. Questa probabilità viene calcolata tramite una regola detta *random proportional rule* e che è rappresentata dalla formula:

$$p_{ij} = \frac{[\tau_{ij}]^\alpha [\eta_{ij}]^\beta}{\Sigma_{l \in N_i} [\tau_{il}]^\alpha [\eta_{il}]^\beta} \tag{1.2}$$

con:

$$j \in N_i \tag{1.3}$$

con η_{ij} che rappresenta l'inverso della distanza fra nodo i e nodo j, α e β che rappresentano l'influenza relativa rispettivamente di τ e η ed infine N_i che rappresenta l'insieme dei vicini raggiungibili dal nodo i.
Utilizzando questa formula la probabilità di scegliere un arco aumenta al crescere dei valori di τ ed η associati ad esso. Il ruolo dei parametri di α e β invece è il seguente: più il valore di α si avvicina a zero e più gli archi con una lunghezza associata minore avranno probabilità più alta di esser scelti. Allo stesso tempo, più il valore di β si avvicina allo zero e più la probabilità di scegliere un arco si basa sulla quantità di feromone. Infine per comporre l'insieme del vicinato raggiungibile N_i si deve far riferimento alla memoria della formica artificiale. Dall'insieme devono infatti essere esclusi tutti i nodi precedentemente attraversati.

Aggiornamento del feromone

Ogni volta che tutte le formiche terminano un percorso di andata, il feromone viene aggiornato. La prima operazione, che viene eseguita su tutti gli archi, riguarda l'evaporazione. La quantità di feromone da sottrarre dipende, come precedentemente menzionato, dal parametro ρ. Se con L rappresentiamo l'insieme degli archi di un grafo l'evaporazione può essere calcolata triamite la formula:

$$\tau_{ij} = (1 - \rho)\tau_{ij}, \forall (i, j) \in L \tag{1.4}$$

Successivamente su tutti gli archi appartenenti ai percorsi scelti dale formiche viene invece applicato un incremento di feromone che può assumere un valore da 0 a $\Delta\tau_{ij}$ sulla base della valutazione della bontà del percorso. Questi incrementi possono essere descritti dalla formula:

$$\tau_{ij} = \tau_{ij} + \Delta\tau_{ij}, \forall (i, j) \in L \tag{1.5}$$

1.3.1 Rank-Based Ant System

In base ai metodi utilizzati per la costruzione del percorso e per l'aggionamento del feromone ed in base alla tipologia di problema che si vuole risolvere con il suo utilizzo, esistono molte varianti di Ant System che contengono diverse euristiche di miglioramento ed estensioni. Fra le varianti più note, possiamo citare: l'*Elitist Ant System*, il *MIN-MAX Ant System* e il *Rank-Based Ant System*. Di queste varianti elencate la più

adatta alla soluzione del problema presentato in questa tesi è la terza.
La particolarità del Rank-Based Ant System risiede nel suo modo di determinare la quantità di feromone da depositare durante il percorso di ritorno. Ogni formica, in questa variante, deposita una quantità di feromone associata ad un rango di appartenenza. Prima di effettuare l'aggiornamento del feromone, tutte le formiche che hanno completato un percorso d'andata vengono ordinate da quella la cui memoria artificiale contiene il percorso a lunghezza minore a quella che ha memorizzato il percorso a lunghezza maggiore. Questo ordinamento determina per ogni formica l'appartenenza ad un rango; più il percorso di una formica è lungo, più sarà basso il rango di appartenenza e meno sarà la quantità di feromone da rilasciare sul percorso di ritorno. In alcuni casi, seguendo regole basate sulla tipologia di problema, si può decidere di non fare depositare nessuna quantità di feromone.
Relativamente a quanto mostrato fino ad ora, le performance degli Ant System, comparate con quelle di altri algoritmi metaeuristici, tendono a peggiorare drammaticamente al crescere del istanza in cui viene utilizzato. Per questo motivo, attualmente può essere utilizzato con ottimi risultati solo su grafi che non superano un certo limite di grandezza.

1.3.2 Dall'ACO su TSP all'ACO sul problema target

Ci sono due differenze principali che ci permettono di distinguere il problema target di questa tesi, dal TSP. La prima riguarda la tipologia di grafo che le formiche devono esplorare. Nell'Ant System sopra presentato, il grafo utilizzato come input dell'algoritmo per la risoluzione del TSP è non orientato e completamente connesso. In figura 1.3 ne troviamo un esempio.

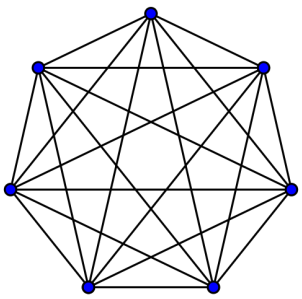

Figura 1.3: Grafo Completo

Nell'algoritmo ACO utilizzato per la risuluzione del problema affrontato in questa tesi invece il grafo è della tipologia mostrata in figura 1.4.

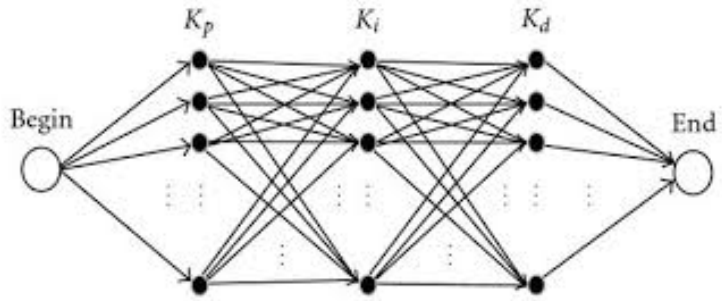

Figura 1.4: Grafo Target

La differenza fra i due grafi sta nel numero di nodi raggiungibili da ogni nodo. Nel primo grafo ogni nodo può raggiungerne qualsiasi altro per mezzo di un singolo arco. Nel grafo target invece ogni nodo può raggiungere un sottoinsieme dei nodi totali del grafo. Questa differenza comporta, in fase di implementazione, qualche accorgimento sulla costruzione e l'utilizzo della matrice dei feromoni che sarà spiegato nel capitolo 3 dedicato allo sviluppo del programma.
La seconda differenza riguarda il costo associato agli archi del grafo. Nel caso del TSP il costo C_{ij}, associato all'arco, equivale alla distanza in km dal nodo i al nodo j. Nel nostro caso invece tale costo è unitario. Questa differenza rende ininfluente il valore del parametro β spiegato in precedenza in quanto utilizzato come influenza dell'inverso del costo di un arco η_{ij}. Di conseguenza la formula per il calcolo della probabilità di scelta di un arco divena quindi:

$$p_{ij} = \frac{[\tau_{ij}]^{\alpha}}{\Sigma_{l \in N_i}[\tau_{il}]^{\alpha}} \tag{1.6}$$

con:

$$j \in N_i \tag{1.7}$$

CAPITOLO 2

Distribuzione di energia elettrica in una rete radiale

In questo capitolo verranno illustrati alcuni concetti base della distribuzione dell'energia nelle reti elettriche, con un particolare focus sulle reti di tipo radiale, per poi passare ad una presentazione semplificata dell'algoritmo Backward Forward Sweep utilizzato per l'analisi di tensioni nodali, correnti sui rami e perdite in linea sulle reti debolmente magliate. Gli argomenti di questo capitolo fanno riferimento a ... per la parte concernente la distribuzione di energia e all'articolo **Power Flow Analysis for Radial Distribution System Using Backward/Forward Sweep Method** [3] per la parte sul Backward Forward Sweep.

2.1 Cenni sulle linee di distribuzione radiali

Le linee di distribuzione di energia elettrica rappresentano l'ultimo miglio dell'intera rete elettrica. Prima di arrivare alla distribuzione si passa per linee ad alta tensione che partono direttamente dalle centrali elettriche e arrivano alla cabine primarie; su queste linee la tensione viene mantenuta ad un valore costante che può essere di 132 o 220 o 380 kV (il valore di corrente varia in base alla richiesta e quindi all'assorbimento dell'intera rete). Nelle cabine primarie, per mezzo di trasformatori detti *abbassatori di tensione*, il valore della tensione viene abbassato 22 o 15 kV. Da qui si arriva alle cabine secondarie. Queste trasformano la tensione, sempre tramite degli abbassatori, da media a bassa, raggiungendo i valori che troviamo comunemente nelle nostre case (400

o 220 V). Da queste cabine secondarie parte la distribuzione capillare. In questa tesi considereremo solamente quest'ultima parte della rete, trattando le cabine secondarie come se fossero generatori e permettendogli di variare il valore di produzione di potenza per renderli più simili a quelli che potrebbero essere degli impianti a energia rinnovabile. La distribuzione capillare, assume spesso una topologia radiale, in figura un esempio di rete di distribuzione radiale.

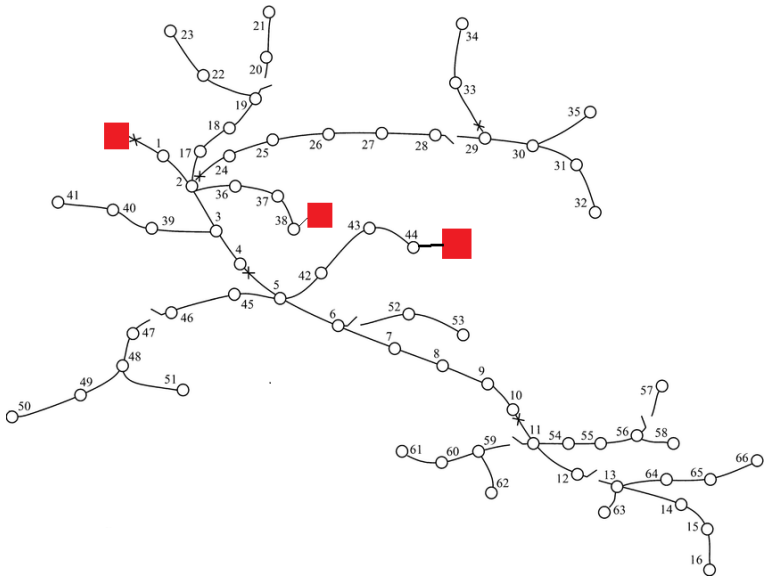

Figura 2.1: Rete Radiale

Nell'illustrazione gli elementi quadrati rappresentano i generatori, mentre tutti gli altri elementi sono carichi. Si può notare che alcune linee possono essere chiuse o aperte; questa possibilità fa variare i valori di assorbimento energetico e di conseguenza la corrente. Variando il passaggio di corrente le perdite possono aumentare o diminuire.

Perdite in linea

Le perdite in linea trattate in questa tesi sono rappresentate dalla potenza dissipata per gli effetti termici dovuti al passaggio di corrente. Il valore di queste perdite dipende in modo statico dalle caratteristiche di resistenza e reattanza delle linee ed in modo dinamico dalla quantità di corrente che le attraversa.
Nella sezione successiva sarà presentato l'algoritmo Backward Forward Sweep e saranno illustrate anche le equazioni che permettono di calcolare le perdite su ogni linea e le perdite totali della rete.

2.2 Il metodo Backward/Forward Sweep

Il backward/forward sweep (BFS) è uno dei metodi per l'analisi del flusso di potenza nei sistemi di distribuzione con topologia radiale, in cui vi sono carichi sbilanciati ed un alto numero di nodi. Gli studi sul flusso di carico ci aiutano a comprendere la natura delle reti di distribuzione esistenti, infatti ci permette di conoscerme le caratteristiche statiche, analizzando i sistemi di alimentazione a regime. Molte delle reti di distribuzione esistenti, per varie cause, fra cui la debole magliatura, carico distribuito sbilanciato, alto rapporto resistenza/reattanza e sistemi multi fase non bilanciati, vengono categorizzate come reti in cattive condizioni. Esistono molti metodi e algoritmi per l'analisi delle reti elettriche; un esempio potrebbe essere il metodo Newton Rapshon, il quale, però, risulta inefficiente nell'analisi delle reti con caratteristiche sopra menzionate, in quanto costruito per l'analisi delle reti fortemente magliate con linee parallele e percorsi ridondanti dai nodi di alimentazione ai nodi di carico. La distribuzione di energia coinvolge innanzitutto il calcolo delle tensioni nodali, da cui ricavare correnti, flussi di potenza e perdite. Il metodo BFS permette di calcolare questi elementi molto velocemente. I sistemi di distribuzione sono l'ultimo miglio delle reti di fornitura elettrica; sono infatti i componenti che consegnano l'elettricità ai consumatori finali, riducendola da media tensione a bassa tensione per mezzo di trasformatori. I sistemi radiali sono i più diffusi in quanto richedono molti meno componenti per la costruzione della rete e risultano per questo meno costosi dei sistemi ad anello centrale. I sistemi radiali inoltre, implicano la non esistenza di cicli nella rete ed ogni nodo è collegato al nodo radice per mezzo di un solo percorso. Le reti di distribuzione radiali sono dunque le meno costose ma le meno affidabili. Il BFS analizza tensioni e perdite su ogni collegamento fra nodi, definendo, per ogni nodo, potenza attiva e potenza reattiva come segue:

$$P_{k+1} = P_k - P_{Loss,k} - P_{Lk+1} \tag{2.1}$$

$$Q_{k+1} = Q_k - Q_{Loss,k} - Q_{Lk+1} \tag{2.2}$$

dove:

P_k è la potenza attiva uscente dal nodo k; Q_k è la potenza reattiva uscente dal nodo k; P_{Lk+1} è la potenza attiva sul nodo k+1; Q_{Lk+1} è la potenza reattiva sul nodo k+1;

Le perdite in linea su ogni collegamento, utilizzando resistenza e reattanza, si possono espiremere come:

$$P_{loss}(k, k+1) = R_k \frac{P_k^2 + Q_k^2}{V_k^2} \tag{2.3}$$

$$Q_{loss}(k, k+1) = X_k \frac{P_k^2 + Q_k^2}{V_k^2} \tag{2.4}$$

dove:

$P_{loss}(k, k+1)$ e $Q_{loss}(k, k+1)$ sono rispetivamente la potenza attiva e reattiva perse sulla linea che congiunge il nodo k e il nodo k+1.

Descriviamo ora in breve il funzionamento del BFS, considerando la sua applicazione su una rete radiale. Il metodo BFS è un processo iterativo che esegue due calcoli importanti ad ogni iterazione. Se consideriamo una rete a sorgente singola, il flusso di carico può essere calcolato iterativamente per mezzo di due set di equazioni ricorsive. Il primo set di equazioni, utilizzato per il calcolo del flusso di potenza sui rami, inizia l'analisi dall'ultimo ramo procedendo all'indietro fino al nodo radice. Il secondo set di equazioni invece, utilizzato per il calcolo delle tensioni ai nodi, inizia l'analisi dal nodo radice e procede verso l'ultimo nodo. La fase di Forward, per il calcolo delle tensioni ai nodi, è un calcolo che si basa sulle variazioni di corrente e flusso di potenza per mezzo di una caduta di tensione. Partendo dal nodo radice, procedendo per livelli (un livello in una rete radiale è rappresentato dall'insieme dei nodi equidistanti dal nodo di alimentazione; con la parola equidistanti non si fa però riferimento alla distanza fisica espressa in metri, ma al numero di nodi presenti ed attraversati dalla corrente fra il nodo radice e il nodo di cui si vuole conoscere il livello, in figura 2.2 un esempio di livellamento) fino a coprire tutti i nodi, questa fase ha lo scopo di calcolare le tensioni su ogni nodo.

La tensione di alimentazione di una sottostazione viene impostata al suo valore corrente e durante la propagazione Forward la potenza effettiva sui rami viene tenuta costante rispetto al valore ottenuto dalla fase di Backward. La fase di Backward calcola il flusso di potenza e le correnti sui rami, variando la tensione. Partendo dal ramo che collega l'ultimo nodo sull'ultimo livello, percorrendo l'albero (rete radiale) fino al nodo radice, la propagazione in Backward calcola la potenza effettiva su ogni ramo considerando la tensione sul nodo della precedente iterazione. Esistono tre varianti del BFS, ognuna differisce dalle altre due per il metodo di calcolo utlizzato per ottenere la

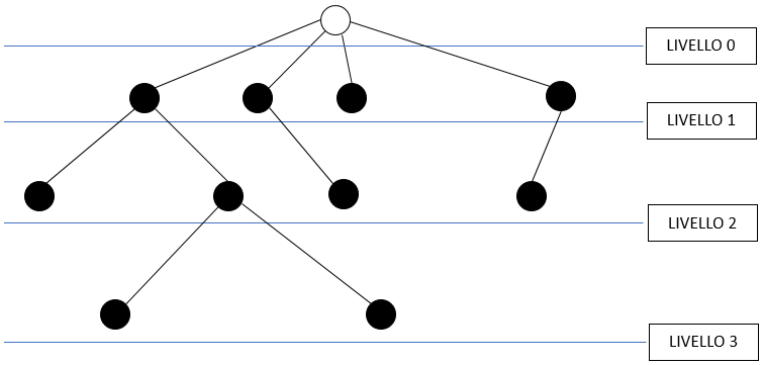

Figura 2.2: Livellamento della rete

quantità elettrica ad ogni iterazione: somma della corrente sulla base di tutte le correnti sul rami; somma della potenza valutata sulla base dei flussi di potenza sui rami; somma delle ammettenze, nella quale per ogni nodo viene valutata l'ammettenza. In pratica le tre varianti del BFS simulano il carico, ad ogni iterazione, con una corrente costante, una potenza costante e un modello di ammettenza costante. Il BFS termina quando viene raggiunto un determinato criterio di convergenza. Il criterio di convergenza può essere raggiunto se ad esempio un valore di tensione calcolato non rispetta un valore di tolleranza preimpostato. Illustriamo ora i calcoli utilizzati nel processo iterativo considerando un nodo k e un nodo adiacente k+1. Le potenze effettive, attiva e reattiva, sul ramo che collega i due nodi possono essere calcolate, in fase di backward, come:

$$P_k = P_{k+1} + R_k \frac{P_{k+1}^2 + Q_{k+1}^2}{V_{k+1}^2} \tag{2.5}$$

$$Q_k = Q_{k+1} + X_k \frac{P_{k+1}^2 + Q_{k+1}^2}{V_{k+1}^2} \tag{2.6}$$

dove:

$$P_{k+1} = P_{k+1} + P_{Lk+1} \tag{2.7}$$

$$Q_{k+1} = Q_{k+1} + Q_{Lk+1} \tag{2.8}$$

P_{Lk+1} e Q_{Lk+1} sono i carichi connessi al nodo k+1; P_{k+1} e Q_{k+1} sono potenza attiva e reativa dal nodo k+1.

Di seguito in figura 2.3 un diagramma di flusso di un algoritmo che esegue il Backword Forward Sweep tratto sempre da [3]:

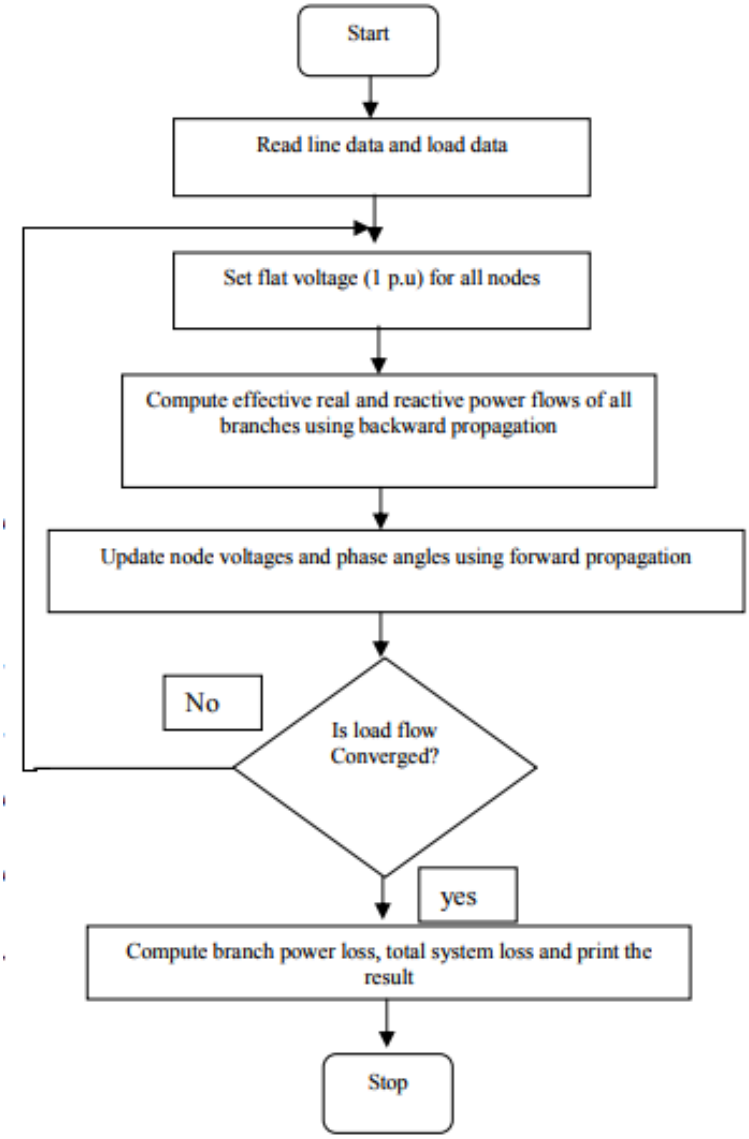

Figura 2.3: Diagramma di flusso BFS

Il BFS analizza le reti considerando il nodo radice unica sorgente della rete. Se, come nel nostro caso target di questa tesi, esistono più sorgenti di energia nella stessa rete radiale, l'algoritmo deve essere eseguito una volta per ognuna delle sorgenti in modo da avere una visione completa di quello che possono essere le configurazioni di correnti, tensioni e perdite. I nodi "generatori" presenti nella rete durante l'analisi del BFS devono essere considerati a potenza attiva negativa. Di seguito un esempio di valori di carico e generazione utilizzati come input dal BFS. Per esprimere la potenza attiva e la potenza reativa di un carico o di un generatore vengono comunemente utilizzati

i numeri complessi. La parte Reale del numero rappresenta la potenza attiva, mentre la parte immaginaria rappresenta la potenza reattiva. Entrambe le grandezze sono espresse in *p.u.* (per unit); questa unità di misura è utilizzata per generalizzare le unità di misura con cui normalmente si esprimono le due potenze, che equivalgono a Watt per la potenza attiva e Voltampere reattivo per la potenza reattiva.

- **Carico**

  $1.3 + 0,4i$ (p.u)

- **Generatore**

  $-2.8 + 0,09i$ (p.u)

Un'ultima considerazione per comprendere l'unità di misura *p.u. I valori "per unit" sono come i valori percentuali riferiti alla corrispondente grandezza base e non moltiplicati per 100. Ad esempio, se si definisce una potenza base di sistema di 1 MVA, una potenza attiva di 600 kW diventa 0.6 p.u., oppure una potenza reattiva di 400 kvar diventa 0.4 p.u.* Nel caso specifico di questa tesi, la potenza base di sistema viene impostata al valore di produzione massimo della sorgente di energia che si sta analizzando e viene modificato per ogni esecuzione del BFS.

CAPITOLO 3

Implementazione

Le rete Internet offre, a giorno d'oggi, una copertura globale, rendendo quindi possibile la comunicazione fra stazioni sparse in tutto il mondo. Qualsiasi dispositivo, sia esso di misura o di calcolo, atto alla comunicazione o alla memorizzazione dei dati, può condividere con tutti gli altri le informazioni in suo possesso. Questa architettura di comunicazione globalizzata permette alle persone di partecipare o gestire un insieme di servizi di qualsiasi natura. Per mezzo di intelligenze semplici come quelle dei computer, è possibile automatizzare l'erogazione dei servizi sulla base di una configurazione iniziale ben dimensionata. Si consideri come esempio uno strumento di misurazione e memorizzazione dell'energia consumata da un'utenza che raccolga i dati periodicamente e li memorizzi in un file che rispetta un formato standard per l'elaborazione come il CSV (comma-separated values).
Esistono dispositivi che raccolgono i dati di un contatore e costruiscono una tabella come quella mostrata in figura.

31/01/2015 16:00	0,267	0,081
31/01/2015 16:02	0,297	0,083
31/01/2015 16:04	0,51	0,1

Figura 3.1: Zeus Data

Il primo campo rappresenta il **timestamp** di raccolta dati, il secondo e il terzo campo rappresentano invece rispettivamente la **potenza attiva** e la **potenza reattiva** dell'istante di campionatura. La memorizzazione di questi dati è istantanea e resa disponibile, un sistema di elaborazione può consumare questi e i dati di molti altri

dispositivi e fronire informazioni più specifiche e generalizzate a livello di rete elettrica, per mezzo di algoritmi di analisi come il Backward Forward Sweep descritto nel capitolo precedente.

3.1 Architetture

In questa sezione saranno descritte due architetture di comunicazione ed elaborazione dei dati. Rispettivamente l'architettura reale generalizzata, che si dovrebbe realizzare, e l'architettura utilizzata per la simulazione con particolari più specifici sugli strumenti utilizzati.

3.1.1 Architettura reale

L'architettura di un sistema di raccolta ed elaborazione dati può essere descritta come in figura 3.2.

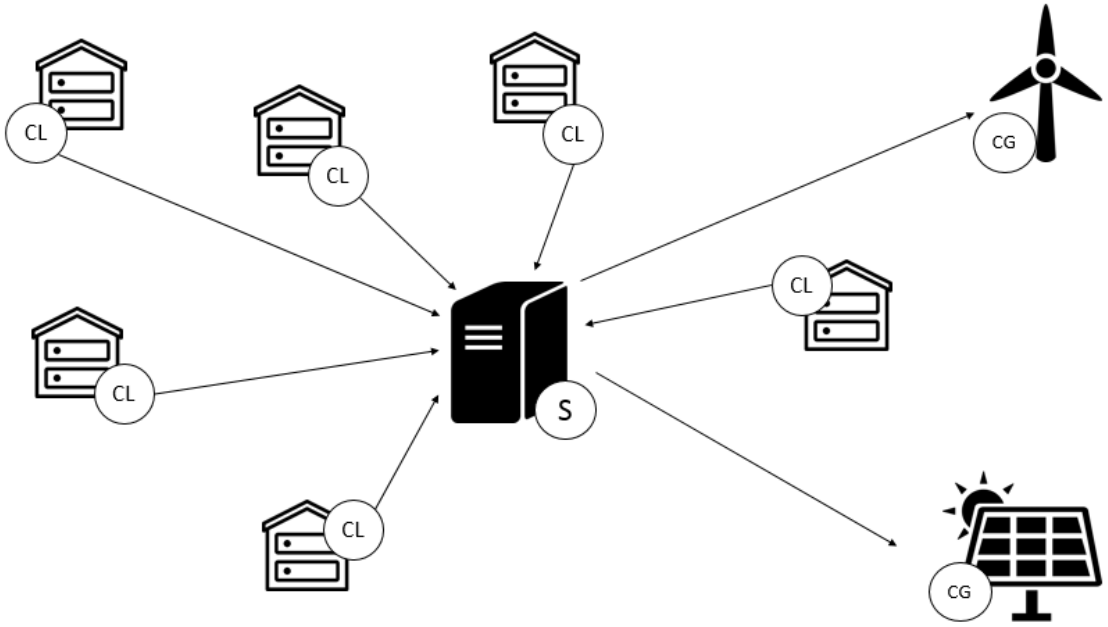

Figura 3.2: Architettura reale

L'illustrazione rappresenta un sistema client-server in cui possiamo notare un server centrale (S) in cui i dati raccolti dai client di "carico" (CL) vengono elaborati e trasformati in profili di generazione inviati ai client di "generazione" (CG). Anche questi

ultimi forniscono un dato istantaneo al server, che esprime il massimo valore di potenza erogabile (trattandosi di sistemi a energia rinnovabile è molto probabile che questo valore cambi nel tempo). Descriviamo nel dettaglio le capacità e le mansioni di ogni attore dell'architettura.

- **Client di Carico**

  In una rete di distribuzione di energia i client di carico sono gli elementi più diffusi e rappresentano un qualsiasi punto di consumo di energia con una potenza attiva e una potenza reattiva associate. Questi client utilizzano un sistema di raccolta dati per mezzo di una lettura sui segnali del contatore e e li rendono disponibili al server centrale per mezzo di una connessione internet e di un semplice programma che gestisce il collegamento (si potrebbe pensare ad un Web Service REST). Gli unici compiti di questo componente sono quindi la raccolta dati e l'invio degli stessi al server.

- **Client di Generazione**

  I client di generazione sono invece degli attuatori; ricevono dal server centrale il profilo di generazione calcolato sulla base dei dati raccolti dai client di carico e da loro stessi ed indicano al generatore vero e proprio il quantitativo di potenza richiesto per soddisfare il carico. Anche in questo caso la connessione con il server centrale può essere gestita da un semplice programma che interroga perodicamente una base dati contenente i valori di attuazione.

- **Server Centrale**

  Sono del server centrale la maggior parte delle mansioni in questo sistema. Esso deve temporizzare la comunicazione con i client di tutti i tipi ed elaborare velocemente i dati raccolti. Il server deve contenere al suo interno un programma che esegua più istanze di Backward Forward Sweep associate ad un uguale numero di Ant Colony Optimization. La potenza di calcolo di questo componente non può dunque essere tralasciata in quanto necessita di risorse crescenti sulla base della quantità di client da servire.

In Figura 3.2 i client di generazione sono rappresentati da impianti di produzione di energia fotovoltaici o eolici; questo perchè gli attuali sistemi di distribuzione di energia seguono un'architettura in cui la produzione è centralizzata (si pensi alle centrali

elettriche) e non distribuita; più ci si avvicina ai consumatori (carichi) finali e più tensioni e correnti, provenienti dalle centrali, vengono dimensionate per essere gestite delle torrette di distribuzione. In questo sistema possiamo quindi escludere le torrette di distribuzione correntemente in uso oppure considerarle come altri client di generazione.

3.1.2 Architettura per la simulazione

L'archiutettura utilizzata per la simulazione risulta molto più specifica ed è qui rappresentata.

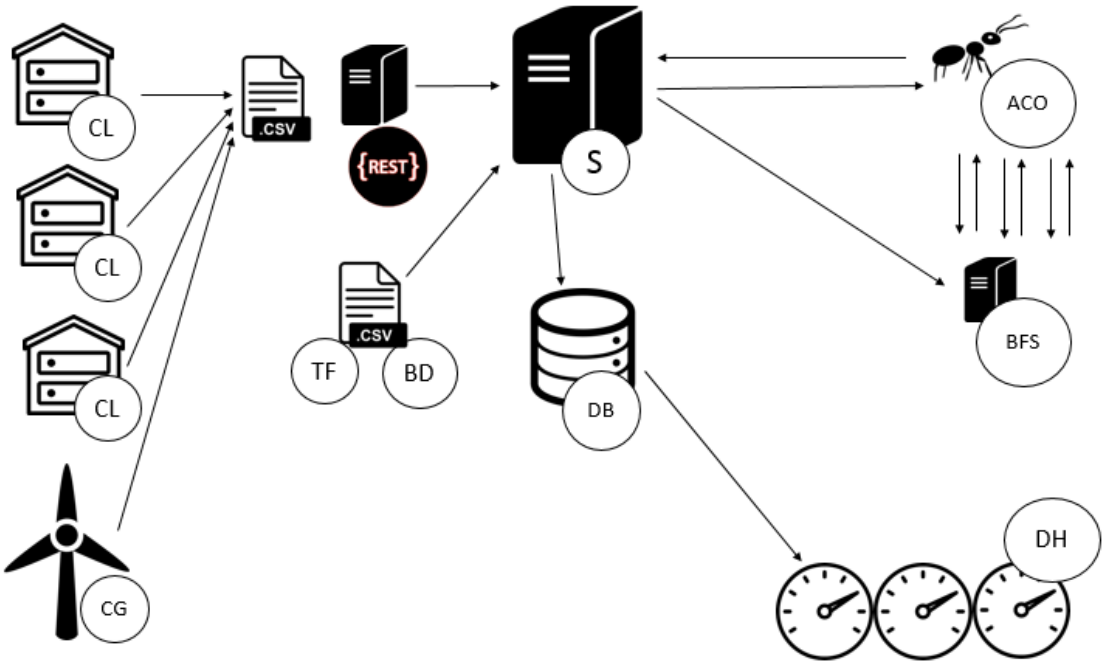

Figura 3.3: Architettura per la simulazione

In Figura 3.3 possiamo notare dei componenti comuni all'architettura reale, come i client di carico (CL) e di generazione (CG) e il server centrale (S). Gli altri comonenti rappresentano un file CSV, un REST service, un database (DB), un Ant Colony (ACO) a stretto contatto con un Backward Forward Sweep (BFS), altri file CSV (TF e BD) ed infine una Dashboard (DH).
Di seguito sono una breve spiegazione delle mansioni di ogni componente in relazione all'architettura reale.

- **Client di carico e di generazione**

In questa simulazione la potenza attiva e reattiva di ogni client di carico è rappresentata da un file csv che contiene anche un valore di potenza per ogni client di generazione che indica la massima potenza erogabile da quel generatore.

- **Servizio REST**

  Un semplice programma espone i dati del file CSV tramite un REST Service e li rende disponibili al server centrale.

- **File CSV**

  I due file di CSV rappresentano rispettivamente le topologie di rete analizzate (TF) e i dati di resistenza e reattanza sui rami reali che collegano carichi e generatori nella topologia di rete (BD).

- **Server Centrale**

  Il server centrale, in questo caso, preleva i dati di carico e generazione, collegandosi al servizio REST in http, legge uno o più file CSV contenenti la topologia di rete; per mezzo di queste informazioni costrusce le strutture dati da inviare all'Ant Colony e al Backward Forward Sweep per l'elaborazione. Riceve dai due algoritmi sopracitati il risultato del problema di minimizzazione delle perdite in linea e le memorizza su un database.

- **Ant Colony**

  E' l'algoritmo principale del sistema ed è utilizzato per l'esplorazione delle configurazioni di generazione possibili sulla topologia di rete eletrica.

- **Backward Forward Sweep**

  E' l'algoritmo utilizzato per l'analisi delle di correnti sui rami, tensioni nodali e perdite sulla rete elettrica. Questa operazione viene eseguita per tutti i generatori presenti nella rete.

- **Database**

  Questo componente memorizza i risultati dell'elaborazione e li rende disponibili per la Dashboard.

- **Dashboard**

  La Dashboard è una semplice pagina http che preleva i dati del database e li rende interpretabili.

3.2 Implementazione dei componenti

In questa sezione verranno illustrate le implementazioni di ugnuno dei componenti elencati precedentemente. Tutti i componenti sono stati sviluppati in linguaggio C# utilizzando template .NET disponibili su Visual Studio.

3.2.1 Il servizio REST

Il servizio REST implementato rende disponibili al server centrale i dati presenti sul file csv.
Per la realizzazione di questo semplice Web Service è stato utilizzato un template "ASP.NET Web Application" disponibile in .NET. Il template costruisce automaticamente un interfaccia ed una classe che la implementa, su cui è molto semplice apportare modifiche per creare il servizio di cui si ha bisogno.
Di seguito rispettivamente l'interfaccia e la classe di implementazione.

Listing 3.1: REST interface

```
[ServiceContract]
public interface IRestServiceImpl
{
  [OperationContract]
  [WebInvoke(Method = "GET",
        ResponseFormat = WebMessageFormat.Json,
        BodyStyle = WebMessageBodyStyle.Wrapped,
        UriTemplate = "json/{id}")]
  string JSONData(string id);
}
```

Listing 3.2: REST class

```
public class RestServiceImpl : IRestServiceImpl
{
 String[] values;
 public RestServiceImpl ()
 {
  char[] separators = new char[2];
  separators[0] = ';';
  separators[1] = '\n';
  values = File.ReadAllText(@"zeus1.csv").Split(separators);
 }

 public string JSONData(string id)
 {
   String[] values;
   char[] separators = new char[2];
   separators[0] = ';';
   separators[1] = '\n';
   values = File.ReadAllText(@"zeus"+id+".csv").Split(separators);
   string datetime = values[0].Replace('/', '-');
   string result = values[1] + " " + values[2];
   res = result.Replace("\r\n", "").Replace("\r", "").Replace("\n", "");
   var lines = File.ReadAllLines(@"zeus"+id+".csv");
   File.WriteAllLines(@"zeus"+id+".csv", lines.Skip(1).ToArray());
   return res;
  }
}
```

Il servizio risponde all'url:

`http://localhost:65000/RestServiceImpl.svc/json/{id}`

in cui il parametro finale {id} indica il file da cui estrarre i dati.
Nell'architettura reale i dati dovrebbero essere raccolti collegandosi singolarmente ad ogni nodo di carico e di generazione. Nell'architettura simulata, non avendo un numero di nodi partecipanti variabile a disposizione, tutti i dati risiedono nello stesso file. Per completare l'analisi, il BFS deve eseguire i calcoli escludendo, uno per volta, tutti i client di generazione; l'id del file, passato come parametro al servizio REST, seleziona il file contenente i dati ordinati in modo che il nodo di generazione escluso sia il primo

valore.
Il servizio restituisce i dati in formato JSON come segue:

```
{"JSONDataResult":"0,5 0,3"}
```

in cui il primo valore indica la potenza attiva ed il secondo la potenza reattiva.

3.2.2 Topologia e dati sui rami

Il server deve eseguire l'analisi escludendo un nodo di generazione alla volta. Per questo, si ha bisogno di un file di topologia che contenga una matrice di adiacenza nodo-nodo e le informazioni necessarie per generare tutte le matrici di adiacenza scambiando riga e colonna dei nodi esclusi con il nodo 0, alternativamente. In questo caso si parla di topologia di rete elettrica; non bisogna quindi confondersi con la topologia di rete di comunicazione indicata nella sezione delle architetture.
Di seguito un esempio di topologia a 3 generatori e 3 carichi per comprendere meglio quali siano i dati, sui file testuali, in ingresso al server.
Si consideri la topologia in figura 3.4.

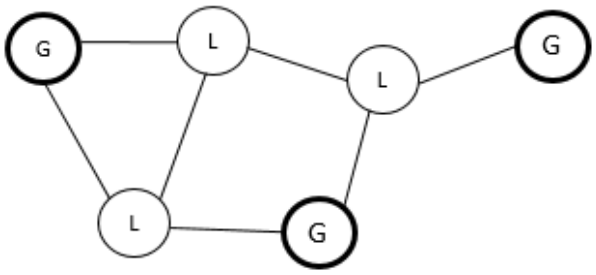

Figura 3.4: Topologia Generatori-Carichi

I generatori (G) vengono assegnati al nodo 0 alternativamente, mentre i carchi (L) mantengono la loro numerazione. Ad ogni configurazione con nodo 0 assegnato corrisponde una matrice di adiacenza differente.

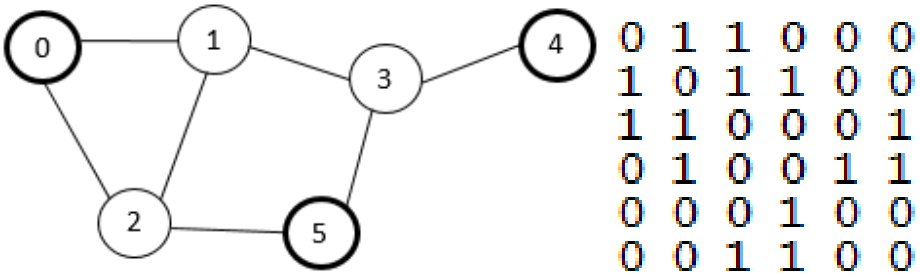

Figura 3.5: Topologia Generatori-Carichi nodo 0

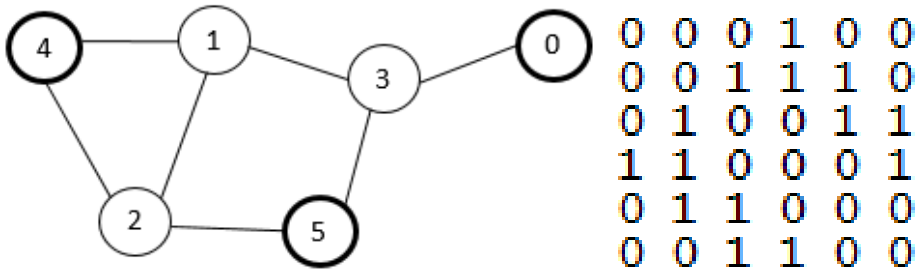

Figura 3.6: Topologia Generatori-Carichi nodo 4

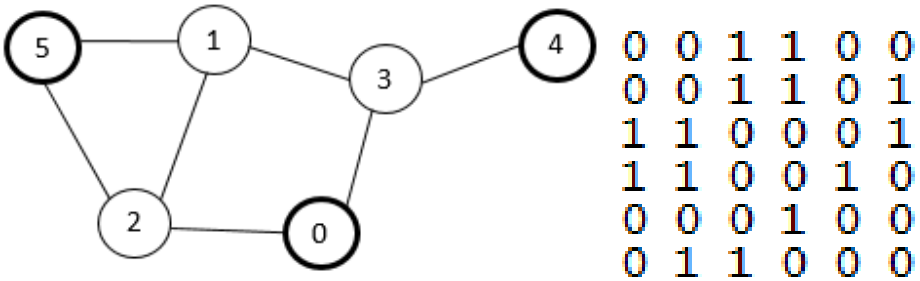

Figura 3.7: Topologia Generatori-Carichi nodo 5

All'inizio del file mostrato in figura 3.8 si possono notare 2 numeri iniziali che indicano rispettivamente il numero di generatori e il numero di nodi totali nella topologia; nella seconda riga del file invece ci sono gli indici dei nodi generatori della rete.
Oltre alle informazioni sulla topologia, per eseguire i calcoli su correnti sui rami e tensioni nodali, il server ha bisogno di conoscere i valori di resistenza R e reattanza X associati ad ogni ramo. Un secondo file descrive questi due valori associandoli ad un arco come in figura 3.9.

3	6				
0	4	5			
0	1	1	0	0	0
1	0	1	1	0	0
1	1	0	0	0	1
0	1	0	0	1	1
0	0	0	1	0	0
0	0	1	1	0	0

Figura 3.8: Topologia 6 Nodi 3GEN

	7	0_1	0_2	1_2	1_3	2_5	3_4	3_5
R		0.007820	0.015640	0.011730	0.007820	0.015640	0.011730	0.015640
X		0.002120	0.004240	0.003180	0.002120	0.004240	0.003180	0.004240
	7	3_1	3_2	1_2	1_0	2_5	0_4	0_5
R		0.007820	0.015640	0.011730	0.007820	0.015640	0.011730	0.015640
X		0.002120	0.004240	0.003180	0.002120	0.004240	0.003180	0.004240
	7	5_1	5_2	1_2	1_3	2_0	3_4	3_0
R		0.007820	0.015640	0.011730	0.007820	0.015640	0.011730	0.015640
X		0.002120	0.004240	0.003180	0.002120	0.004240	0.003180	0.004240

Figura 3.9: Dati sui rami 6 Nodi

3.2.3 Server Centrale

In questa simulazione sul server centrale è presente un programma (ACO Losses Finder) che permette di gestire tutte le operazioni tramite un semplice Form. Il programma è stato svilppato in Visual Studio utilizzando il template "Windows Form" di .NET.

Tramite questo form (figura 3.10) è possibile eseguire le seguenti operazioni:

- **Reboot Topology**

  Il button "Reboot Topology" permette di caricare i dati dai file, creare le strutture dati e calcolare tutti gli alberi ricoprenti della topologia;

- **Selezione Nodo 0**

  Al centro del Form, in una listbox, sarà possibile selezionare un numero di topologie pari al numero di generatori nella rete, assegnando il nodo zero ad uno dei nodi generatori;

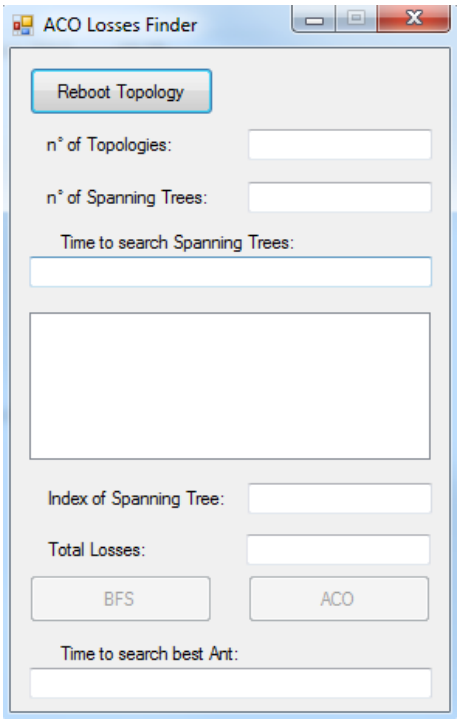

Figura 3.10: ACO Losses Finder

- **BFS**

  Il button "BFS" inizilizza l'algoritmo Backward Forward Sweep istanziando un thread differente per ogni albero ricoprente trovato e mettendosi in attesa di una connessione TCP da parte di un client;

- **ACO**

  Il button "ACO", da utilizzare solo successivamente all'inizializzazione del BFS, permette di avviare l'esplorazione delle formiche istanziando un thread differente per ogni albero ricoprente trovato e inviando al BFS richieste di calcolo, tramite una connessione TCP;

L'operazione di **Selezione Nodo 0** può essere automatizzata da file di configurazione in modo da far completare l'analisi totale senza l'intervento interattivo.

Caricamento delle strutture dati per gli alberi ricoprenti

Il programma permette l'analisi di topologie di rete radiali, utilizzando come input i due file descritti nella sezione 3.2.2. Una classe statica espone il metodo *allSpanning-*

Trees(...) che legge i file di topologia e dati sui rami e produce due liste che contengono rispettivamente la lista di tutti gli alberi ricoprenti, replicati per ogni matrice di adiacenza, e una lista contenente le resistenze R e le reattanze X sui rami.
La prima operazione eseguita dal metodo *allSpanningTrees(...)* è la costruzione di tutte le matrici di adiacenza, una per ogni generatore nella topologia, tramite la funzione *calculateLMatrixAutomatic(...)* che prende in input il nome del file su cui risiede la matrice di adiacenza nodo-nodo dal punto di vista del nodo 0 e restituisce una lista di array multidimensionali di numeri interi che rappresenta l'insieme di tutte le matrici di adiacenza.

Listing 3.3: calculateLMatrixAutomatic(..)

```
private static List<int[,]> calculateLMatrixAutomatic(
  string generalTopology, out int [] gen_indexes)
{
  List<int[,]> graphs = new List<int[,]>();
  int generator_count = 0;
  int nodes = 0;
  string[] lines = System.IO.File.ReadAllLines(generalTopology);
  gen_indexes = null;
  if (lines.Length == 0)
    return null;
  string[] gen_and_nodes = lines[0].Split(';');
  generator_count = int.Parse((gen_and_nodes[0]
                .Replace(';', ' ')).Trim());
  nodes = int.Parse((gen_and_nodes[1]
                .Replace(';', ' ')).Trim());
  string[] generators_nodes_strings = lines[1].Split(';');
  int[] generators_nodes = new int[generator_count];
  for (int i = 0; i < generator_count; i++) {
    generators_nodes[i] = int.Parse(generators_nodes_strings[i]
        .Replace(';',' ').Trim());
  }
  gen_indexes = new int[generators_nodes.Length];
  generators_nodes.CopyTo(gen_indexes, 0);
  // lettura della topology dal nodo 0
  int[,] graph = new int[nodes, nodes];
  for (int i = 2; i < nodes + 2; i++) {
        string[] lineValues = lines[i].Split(';');
```

```
  for (int j = 0; j < lineValues.Length; j++) {
        string value = lineValues[j].Replace(';', ' ').Trim();
    graph[i-2, j] = int.Parse(value);
    }
  }
  graphs.Add(graph);
  int[] zero_row_column = new int [nodes];
  for (int i = 0; i < nodes; i++)
        zero_row_column[i] = graph[0,i];
  for (int i = 1; i < generators_nodes.Length; i++) {
        int[,] target_graph = (int[,])graph.Clone();
        int target_node = generators_nodes[i];
        int[] target_row_column = new int[nodes];
  for (int j = 0; j < nodes; j++)
        target_row_column[j] = graph[target_node, j];
  // scambio righe e colonne
  for (int j = 0; j < nodes; j++) {
        target_graph[0, j] = target_row_column[j];
        target_graph[j, 0] = target_row_column[j];
        target_graph[target_node ,j] = zero_row_column[j];
        target_graph[j,target_node] = zero_row_column[j];
  }
  graphs.Add(target_graph);
  }
  return graphs;
}
```

Successivamente, per ogni grafo di adiacenza restituito, viene eseguita una ricerca di tutti gli alberi ricoprenti relativi ad esse e vengono caricati in una lista. Questa operazione è stata realizzata tramite molti metodi che semplificano la topologia in modo da rendere più leggera la ricerca di tutti gli alberi ricoprenti.
Una prima funzione *getAdiacenze(...)* prende il grafo in input e restituisce una lisa di Tuple di *int* che rappresenta tutte le adiacenze.

Listing 3.4: getAdiacenze(..)

```
private static List<Tuple<int, int>> getAdiacenze(
        int[,] graph, int nodes)
{
 var adiacenze = new List<Tuple<int, int>>();
```

```
 for (int i = 0; i < nodes; i++) {
   for (int j = 0; j < nodes; j++) {
     if (graph[i, j] == 1) {
        var tuple = new Tuple<int, int>(i, j);
        adiacenze.Add(tuple);
     }
   }
 }
 return adiacenze;
}
```

A questo punto due metodi, *semplificaAdiacenze(...)* ed *eliminaSemplificate(...)*, eliminano dalla lista le adiacenze ridondanti (es: (1,2),(2,1)) trovate nel grafo (il grafo analizzato è una matrice quadrata simmetrica rispetto alla diagonale, perciò per ogni adiacenza ci saranno 2 entry nella matrice).
Segue il caricamento dei dati sui rami presi dal secondo file CSV descritto nella sezione 3.2.2. Questi dati vengono caricati in array multidimensionale di Tuple di *double* in cui ogni Tupla rappresenta i valori di Resistenza R e Reattanza X e gli indici dell'array rappresentano l'arco relativo.
Una ulteriore semplificazione viene elaborata dal metodo *Ovvie(...)* analizzano tutte le adiacenze trovando quegli archi che nel grafo risultano semplificabili in quanto obbligatori. Se un nodo è connesso al resto della rete da un solo arco, allora questo è semplificabile ai fini della ricerca degli alberi ricoprenti. Questi archi vengono eliminati per semplificare la ricerca degli alberi ricoprenti, ma verrranno reinseriti nel grafo prima dell'inizio dell'elaborazione di ACO e BFS. La funzione prende in input la lista di adiacenze e restituisce le adiacenze da eliminare momentaneamente.

Listing 3.5: Ovvie(..)

```
public static List<Tuple<int, int>> Ovvie(
        List<Tuple<int, int>> adiacenzeSemplificate, int nodes)
{
  List<Tuple<int, int>> adiacenzeNonOvvie = new List<Tuple<int, int>>();
  int[] support = new int[nodes];
  List<Tuple<int, int>> adiacenzeDaOvviare = new List<Tuple<int, int>>();
  for (int i = 0; i < nodes; i++) {
    foreach (Tuple<int, int> x in adiacenzeSemplificate) {
      if (x.Item1 == i || x.Item2 == i)
```

```
        support[i]++;
    }
  }
  List<int> indexes = new List<int>();
  for (int i = 0; i < nodes; i++)
    if (support[i] == 1)
          indexes.Add(i);
  foreach (int index in indexes) {
    foreach (Tuple<int, int> x in adiacenzeSemplificate) {
      if (x.Item1 == index || x.Item2 == index)
              adiacenzeDaOvviare.Add(x);
    }
  }
  return adiacenzeDaOvviare;
}
```

L'ultimo metodo, chiamato all'interno della funzione *allSpanningTrees(...)* ed all'interno del ciclo che ripete la costruzione per ogni grafo di adiacenza, è *Combinations(...)* il cui codice è tratto in parte da un sito internet [4]. Questo metodo prende come parametri: numero di archi partecipanti alla ricerca, numero di nodi partecipanti alla ricerca, la lista di adiacenze semplificate, una lista dei nodi partecipanti alla ricerca e la lista delle adiacenze trovate dalla funzione *Ovvie(...)*; ritorna infine una lista di matrici di adiacenza che contiene tutti gli alberi ricoprenti del grafo analizzato.
All'interno del metodo vengono esaminate tutte le combinazioni composte da $n-1$ archi dove n è il numero di nodi. Per ogni combinazione viene chiamato un secondo metodo, *provaCombinazione(...)*, che controlla che l'insieme dagli archi selezionati componga un albero ricoprente. Nel caso in cui l'albero trovato sia ricoprente, verrà aggiunto alla lista di matrici di adiacenza.

Listing 3.6: Combinations(..)

```
private static List<List<Tuple<int, int>>> Combinations(
        int n, int k, List<Tuple<int, int>> adiacenzeSemplificate,
        List<int> nodiPartecipanti, int nodes,
        List<Tuple<int, int>> adiacenzeOvviate)
{
  List<List<Tuple<int, int>>> matriciDiAdiacenza =
  new List<List<Tuple<int, int>>>();
```

```
  List<int[]> ret = new List<int[]>();
  int[] comb = new int[k];
  int[] combMax = new int[k];
  if (n == 0) return null;
  if (k == 0) return null;
  if (n < k) return null;
  for (int i = 0; i < k; i++)
    comb[i] = i;

  provaCombinazione(comb,adiacenzeSemplificate,nodiPartecipanti,
        nodes,adiacenzeOvviate,matriciDiAdiacenza);

  for (int i = 0; i < k; i++)
        combMax[i] = n - k + i;
  while (true) {
        bool trovatoIncrementabile = false;
        int indiceIncrementabile = -1;
        for (int i = k - 1; i >= 0; i--) {
          if (i == k - 1) {
                if (comb[i] < combMax[i]) {
                  trovatoIncrementabile = true;
          indiceIncrementabile = i;
          break; } }
          else {
                if (comb[i] < combMax[i] && comb[i] + 1 != comb[i + 1]) {
                  trovatoIncrementabile = true;
          indiceIncrementabile = i;
          break; } }
    }
        if (!trovatoIncrementabile)
          break;
        comb[indiceIncrementabile]++;
        if (indiceIncrementabile != k - 1) {
          for (int i = indiceIncrementabile + 1; i < k; i++)
                comb[i] = comb[i - 1] + 1; }
        provaCombinazione(...);
  }
  return matriciDiAdiacenza;
}
```

Nel metodo *provaCombinazione(...)* viene chiamata una funzione (*findTrailsFromZero(...)*)che effettua la vera e propria visita dell'albero cercando un percorso che unisca il nodo 0 a tutti gli altri nodi per mezzo di un terzo metodo (*canGoToAllNodes(...)*). La funzione restituisce un valore booleano che indica se aggiungere l'albero analizzato alla lista degli alberi ricoprenti.

Listing 3.7: findTrailsFromZero(..)

```
public static bool findTrailsFromZero(
        List<Tuple<int, int>> tuple, int nodes)
{
  int[,] incidenceMatrix = new int[nodes, nodes];
  int[] nodiFittizzi = new int[nodes];
  for (int i = 0; i < nodes; i++) nodiFittizzi[i] = i;
  List<int> nodiOriginali = new List<int>();
  foreach (Tuple<int, int> t in tuple) {
        if (!nodiOriginali.Contains(t.Item1))
          nodiOriginali.Add(t.Item1);
        if (!nodiOriginali.Contains(t.Item2))
          nodiOriginali.Add(t.Item2);
  }
  nodiOriginali.Sort();
  Hashtable corrispondenze = new Hashtable();
  for(int i = 0; i < nodes; i++)
        corrispondenze.Add(nodiOriginali[i],nodiFittizzi[i]);
  foreach (Tuple<int, int> t in tuple) {
   incidenceMatrix[(int)corrispondenze[t.Item1],
     (int)corrispondenze[t.Item2]] = 1;
   incidenceMatrix[(int)corrispondenze[t.Item2],
     (int)corrispondenze[t.Item1]] = 1;
  }
  return canGoToAllNodes(incidenceMatrix);
}
```

Listing 3.8: canGoToAllNodes(..)

```
public static bool canGoToAllNodes(int[,] m)
{
  int nodes = (int)Math.Sqrt(m.Length);
  int[] nodiDaRaggiungere = new int[nodes];
  nodiDaRaggiungere[0] = 1;
  List<int> nodiAttraversati = new List<int>();
  var vertices = new int[nodes];
  for (int i = 0; i < nodes; i++) vertices[i] = i;
    var e = new List<Tuple<int, int>>();
  int count = 0;
  for (int i = 0; i < nodes; i++) {
        for (int j = 0; j < nodes; j++) {
          if (m[i, j] == 1) {
        count++;
        e.Add(Tuple.Create(i, j)); } }
  }
  var edges = new Tuple<int, int>[e.Count];
  for (int i = 0; i < e.Count; i++)
        edges[i] = e[i];
  var graph = new Graph<int>(vertices, edges);
  var algorithms = new Algorithms();
  string result = string.Join(", ", algorithms.DFS(graph, 1));
  string[] res = result.Split(',');
  if (res.Length < nodes)
    return false;
  return true;
}
```

In quest'ultimo metodo è stata utilizzata una classe il cui codice è pubblicato su internet [5] , che esegue la visita in profondità ricevendo una matrice di adiacenze in input.

Inizializzazione dei BFS

Per la complessità dei calcoli matriciali e per l'utilizzo dei numeri complessi, per questa sezione del codice è stata utilizzata una libreria esterna a .NET disponibile tramite NuGet [6].
Una volta composte le strutture dati contenenti tutti gli alberi ricoprenti relativi alle matrici di adiacenza, calcolate per ogni generatore della rete, tramite il tasto "BFS" del Form vengono inizializzate un numero di istanze dell'algoritmo BFS pari al numero di alberi ricoprenti moltiplicato per il numero di generatori nella rete. Ogni istanza riceve un vettore di carichi e generatori (in forma di numeri complessi espressi in p.u. - vedi sezione 2.2) precedentemente composta leggendo i dati provenienti dal servizio REST. Di seguito i due metodi annidati che leggono i dati interrogando il servizio e compongono il vettore di numeri complessi.

Listing 3.9: readPowersLoads(..)

```
public static Vector readPowersLoads(int n, int selectedTopology)
{
  Vector v = null;
  double[,] readedValues =
        readCurrentPowersLoadsValue(n,selectedTopology);
  v = new DenseVector(n);
  double[,] valuesVector = new double[n, 2];
  valuesVector = readedValues;
  for (int i = 0; i < n; i++)
    v[i] = new Complex(valuesVector[i, 0], valuesVector[i, 1]);
  return v;
}
```

Listing 3.10: readCurrentPowersLoadsValue(..)

```
private static double[,] readCurrentPowersLoadsValue(
                int n, int selectedTopology)
{
  double[,] readedValues = new double[n, 2];
  for (int j = 0; j < n; j++) {
    string sURL;
    sURL = "http://localhost:65000/RestServiceImpl.svc/json/"
                                + selectedTopology;
```

```
    WebRequest wrGETURL;
    wrGETURL = WebRequest.Create(sURL);
    Stream objStream;
    objStream = wrGETURL.GetResponse().GetResponseStream();
    StreamReader objReader = new StreamReader(objStream);
    string sLine = "";
    int i = 0;
    while (sLine != null) {
      i++;
      sLine = objReader.ReadLine();
        if (sLine != null) {
            string line = sLine;
            string[] as = line.Split(':');
            string[] valuesss = asds[1].Split(' ');
            valuesss[0] = valuesss[0]
                        .Replace('"', ' ').Trim();
            valuesss[1] = valuesss[1]
                        .Replace('"', ' ').Replace('}', ' ').Trim();
        readedValues[j, 0] = double.Parse(valuesss[0]);
            readedValues[j, 1] = double.Parse(valuesss[1]);
          }
    }
  }
  return readedValues;
}
```

Utilizzando la classe `System.Threading` vengono quindi avviati i thread che eseguono le istanze BFS.

Listing 3.11: Avvio istanze BFS

```
for (int i = 0; i < numeroListe; i++) {
  int localNum = i;
  Vector v = new DenseVector(this.nodesWithoutSlack);
  int k = 0;
  for (int j = 0; j < this.nodesTotal; j++) {
        if (j != nodo_slack) {
          v[k] = this.Scar[j];
      k++; } }
  Complex nodoSlack = this.Scar[nodo_slack];
  threadsArrayBFS[i] = new Thread(() =>
```

```
                BFSController.BFS_Main(this.listeDiAdiacenza[localNum],
                    this.branchesRX,
                localNum, v, nodoSlack));
  threadsArrayBFS[i].Start();
}
```

Nella sezione di codice qui sopra, ad ogni thread viene associata una funzione statica della classe BFSController, BFS_Main. BFS_Main prende in input una delle liste di adiacenza disponibili, l'array dei dati sui rami, il vettore dei carichi e dei generatori ed un numero complesso che rappresenta il nodo 0 relativo alla lista. Successivamente costruisce una copia della topologia analizzata, utilizzando degli oggetti creati localmente e il metodo *initTopology* della classe statica `BFS_Methods` che restituisce un oggetto di tipo `BFSTopology`. Questo oggetto è molto semplice ed è costituito di una matrice L di adiacenze, un vettore delle resistenze, un vettore delle reattanze e il vettore dei carichi e generatori. Alla fine del metodo viene invocata la funzione di routine vera e propria del BFS, *BFSStart*.

Listing 3.12: BFS_Main

```
public static void BFS_Main(
                List<Tuple<int, int>> lista,
                Tuple<double, double>[,] branchesRX, int index,
                Vector Scar, Complex nodoSlack)
{
  double[] r = new double[lista.Count];
  double[] x = new double[lista.Count];
  int nodes = 0;
  foreach (Tuple<int, int> t in lista) {
        if (t.Item1 > nodes) nodes = t.Item1;
         ...
         Ricostruzione di topologia (Matrice L)
         e dei vettori di resistenze e reattanze (r, x)
         ...
  BFSTopology topo = BFS_Methods.initTopology(L, r, x);
  BFSStart(topo,index,Scar,nodoSlack);
  return;
}
```

La funzione *BFSStart* si mette in ascolto su una porta TCP ed attende una richiesta da parte dell'algoritmo ACO eseguito parallelamente. Il BFS si mette in attesa di una connessione TCP in *loop* per un numero di volte pari al numero di formiche moltiplicato peer il numero di iterazioni dell'ACO (questi dati sono disponibili da file di configurazione). Il codice di questa funzione prende spunto da una funzione Matlab reperita al Politecnico di Torino ed è la soluzione ad una esercitazione che richiedeva di costruire un programma che eseguisse il BFS.

```
%dati linea
n=9; Imax=194;   %A
toll=1e-5;
%matrice delle incidenze
L=-eye(n,n); L(4,1)=1; L(5,1)=1; L(6,2)=1; L(7,3)=1; L(8,2)=1; L(9,3)=1;
gamma=inv(L);
%parametri di linea
R=[0.007820, 0.015640, 0.011730, 0.011730, 0.013685, 0.011730, 0.011730, 0.011730, 0.007820];
X=[0.002120, 0.004240, 0.003180, 0.003180, 0.003710, 0.003180, 0.003180, 0.003180, 0.002120];
%ammettenze trasversali delle linee (vettore)
Ysh=zeros(n,1);
Ytotnod=abs(L')*Ysh/2;  %prendo la colonna corrispondente (abss perchè ho i meno 1)
%matrice delle impedenze
Zb=zeros(n,n);
for i=1:length(R)
    Zb(i,i)=diag([R(i)+j*X(i)]);
end
%Carichi a potenza assegnata
Scar=[0.6+0.4i,0.5+0.3i,0.1+0.09i,0.6+0.4i,1.3+1.1i,1.3+1i,0.1+0.09i,0.8+0.6i,0.3+0.1i];
 %carichi ad ammettenza assegnata
 Ycar=zeros(n,1);
 %Grandezze base
Vbase=10;                                  %kV
Sbase=1000;                               %kVA
Ibase=Sbase/(sqrt(3)*Vbase); %A
%Limiti termici linee
Ilim_A=194;                               %A;
Imax=Ilim_A/Ibase*ones(n,1);
%Tensione nodo slack
V0=1;
V0vett=V0*ones(n,1);
tolleranza=1e-6;
errore=inf;
v=V0vett;
is=zeros(n,1);      %correnti nodali
ib=zeros(n,1);      %correnti dei rami
%Processo iterativo
iter=0;                 %contatore del numero di iterazioni
```

Figura 3.11: Matlab BFS Initialization

Di seguito la parte di inizializzazione del BFS, tradotta in C#, sulla base dei dati ricevuti in fase di istanziamento (matrice di adiacenza, dati sui rami).

Listing 3.13: BFSStart

```
public static void BFSStart(BFSTopology t, int index,
                Vector Scar, Complex nodoSlack)
{
  BFSListener listener;
  listener = new BFSListener(index);
  listener.BFSStartListen();
  #region INIT BFS
  int n = t.getNodesNumber();
  double[] r = t.getR();
  double[] x = t.getX();
  int[,] l = t.getL();
  if (Scar == null) {
    log.Error("Impossible to read Powers Loads Data.");
        return;
  }
  t.setScar(Scar);
  Complex[] Rtemp = new Complex[n];
  Complex[] Xtemp = new Complex[n];
  for (int i = 0; i < n; i++) {
    Rtemp[i] = new Complex(r[i], 0);
    Xtemp[i] = new Complex(x[i], 0);
  }
  Complex[,] Ltemp = BFS_LinearAlgebraOperations.eyeMatrix(n);
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      Ltemp[i, j] = new Complex((double)l[i, j], 0); } }
  Complex[] LvectorTemp = new Complex[n * n];
  int k = 0;
  for (int i = 0; i < n; i++) {
    for (int j = 0; j < n; j++) {
      LvectorTemp[k] = Ltemp[j, i]; k++; } }
  BFS_inputSet input = new BFS_inputSet(n, Rtemp, Xtemp, LvectorTemp);
  Matrix gamma = (Matrix)input.getMatrixL().Inverse();
  Vector Ysh = BFS_LinearAlgebraOperations.Zeros(n);
  Matrix LAbs = BFS_LinearAlgebraOperations
```

```
      .absMatrixL(input.getMatrixL());
Vector Ytotnod = (Vector)LAbs.TransposeThisAndMultiply(Ysh);
Matrix Zb = BFS_LinearAlgebraOperations.Zeros(n, n);
for (int i = 0; i < n; i++)
  Zb[i, i] = new Complex(input.getVectorR()[i].Real,
    input.getVectorX()[i].Real);
Vector Ycar = BFS_LinearAlgebraOperations.Zeros(n);
Complex Vbase = 10;
Complex Sbase = 1000;//nodoSlack.Real * (-1) * 1000; // kVA
Complex Ibase = Sbase / (Math.Sqrt(3) * Vbase);
Complex Ilim_A = 194;
Complex[] ones = new Complex[n];
for (int i = 0; i < n; i++) { ones[i] = 1; }
Vector Imax = Ilim_A / Ibase * new DenseVector(ones);
Complex V0 = 1;
Vector V0vett = V0 * new DenseVector(ones);
//tolleranza
double tolleranza = 1e-6;
//errore
double errore = double.PositiveInfinity;
Vector v = V0vett;
Vector Is = BFS_LinearAlgebraOperations.Zeros(n);//correnti nodali
Vector Ib = BFS_LinearAlgebraOperations.Zeros(n);//correnti dei rami
#endregion
int ants_counter =
int.Parse(ConfigurationManager.AppSettings["numberOfAnts"].ToString());
int iteration_counter =
int.Parse(ConfigurationManager.AppSettings["numberOfIterations"].
   ToString());
int counter = (ants_counter * iteration_counter) + ants_counter;
Console.WriteLine("Listening on: {0} - port: {1}",
      listener.ipEndPoint.Address, listener.ipEndPoint.Port);
while (counter > 0) {
 // punto di ingresso dell'ACO
 Console.WriteLine("Listening on: {0} - port: {1}",
 listener.ipEndPoint.Address, listener.ipEndPoint.Port);
 using (TcpClient currentAnt = listener.bfsListener.AcceptTcpClient())
 {
   Vector received_Scar = null;
```

```
NetworkStream stream = null;
try {
     byte[] bytes = new byte[256];
     string data = null;
     data = null;
     stream = currentAnt.GetStream();
     int i;
     while ((i = stream.Read(bytes, 0, bytes.Length)) != 0) {
       data =
             System.Text.Encoding.UTF8.GetString(bytes, 0, i);
             break;
     }
     received_Scar = BFS_Methods
                  .PopolateReceivedScarValues(data, n);
     Vector tempScar = t.getScar();
     for (int j = 0; j < t.getNodesNumber(); j++) {
       if (received_Scar[j] != 0)
           tempScar[j] = received_Scar[j];
     }
     t.setScar(tempScar);
     Scar = t.getScar();
     int iter = 0;
     Vector viniziale;
     errore = Double.PositiveInfinity;
     while (errore > tolleranza) {
                  ...
                  Loop del BFS in attesa di dati dall'ACO
                  ...
     }
     double[] abs_Ib =
                  BFS_LinearAlgebraOperations.normalizeVector(Ib);
     double[] abs_Ib_quadro =
                  BFS_LinearAlgebraOperations.Vector2Power(abs_Ib);
     double[,] realZb =
                  BFS_LinearAlgebraOperations.
                     fromComplexToRealMatrix(Zb);
     Matrix realZb_Complex =
                  (Matrix)BFS_LinearAlgebraOperations
                          .convertMatrixFromDoubleToComplex(realZb)
```

```
                                ;
          Vector abs_Ib_quadro_Complex =
                        (Vector)BFS_LinearAlgebraOperations
                                .convertVectorFromDoubleToComplex(
                                   abs_Ib_quadro);
          Matrix realZb_Complex_Trasp =
                        (Matrix)realZb_Complex.Transpose();
          Vector perdite =
                        (Vector)realZb_Complex_Trasp
                                .TransposeThisAndMultiply(
                                   abs_Ib_quadro_Complex);
          Complex perdite_totali = perdite.Sum();
          double perdite_totali_double = perdite_totali.Real;
          double coef =
                        BFS_Methods.CalcCoef(perdite_totali_double);
          byte[] coef_byte =
                        System.Text.Encoding.UTF8.GetBytes(coef.ToString
                            ());
          stream.Write(coef_byte, 0, coef_byte.Length);
          stream.Close();
          currentAnt.Close();
        }
        catch (SocketException ex) {
          log.Error(ex.Message);
          stream.Close();
          if (currentAnt.Connected)
                  currentAnt.Close();
          continue;
        }
    }
    counter--;
  }
  listener.bfsListener.Stop();
}
```

Al fondo di *BFSStart* si può vedere una fase di composizione del risultato ed una fase di invio del valore ottenuto all'ACO richiedente.

Loop dei BFS

Di seguito il *loop* del BFS descritto nella sezione 2.2, nella sua versione Matlab e C#.

```
while(errore>tolleranza)
    iter=iter+1;
    viniziale=v;     %per il criterio di arresto
    %fase backward
    is=v.*Ycar+Ytotnod.*v+conj(Scar)./conj(v);
    ib=gamma'*is;
    %fase forward
    v=V0vett-gamma*Zb*ib;
    %test di convergenza
    errore=max(abs((v-viniziale))./abs(v));
end

perdite=real(Zb)*abs(ib).^2;
perdite_tot=sum(perdite);
livello=abs(ib)./Imax;
[livellomax,dove]=max(livello);
subplot(1,2,1), plot(abs(v))
subplot(1,2,2), plot(abs(ib))
```

Figura 3.12: Matlab BFS Loop

Listing 3.14: BFSStart Loop

```
while (errore > tolleranza) {
  iter++;
  viniziale = v;
  //fase di backward
  Vector v_Mul_Ycar = (Vector)DenseVector
        .op_DotMultiply(v, Ycar);
  Vector Ytotnod_Mul_v = (Vector)DenseVector
        .op_DotMultiply(Ytotnod, v);
  Vector conjScar_Div_conjv = (Vector)DenseVector
        .op_DotDivide((Vector)Scar
          .Conjugate(), (Vector)v.Conjugate());
  Is = (Vector)(v_Mul_Ycar + Ytotnod_Mul_v + conjScar_Div_conjv);
  Ib = (Vector)gamma.TransposeThisAndMultiply(Is);
  // fase di forward
  Matrix ZbTrasp = (Matrix)Zb.Transpose();
  Matrix gamma_Mul_Zb = (Matrix)gamma.TransposeAndMultiply(Zb);
  Matrix gamma_Mul_Zb_Trasp = (Matrix)gamma_Mul_Zb.Transpose();
```

```
  Vector gamma_Mul_Zb_Mul_Ib = (Vector)gamma_Mul_Zb_Trasp
        .TransposeThisAndMultiply(Ib);
  v = (Vector)(V0vett - gamma_Mul_Zb_Mul_Ib);
  //test di convergenza
  Vector v_Sub_viniziale = (Vector)(v - viniziale);
  double[] abs_v = new double[n];
  double[] abs_v_Sub_viniziale = new double[n];
  abs_v = BFS_LinearAlgebraOperations.normalizeVector(v);
  abs_v_Sub_viniziale = BFS_LinearAlgebraOperations
    .normalizeVector(v_Sub_viniziale);
  errore = (BFS_LinearAlgebraOperations
    .divideDotVectors(abs_v_Sub_viniziale, abs_v)).Max();
}
```

Ad ogni esecuzione del BFS, sotto richiesta dell'ACO e sulla base della configurazione dei profili di generazione inviati dallo stesso, viene restituito un valore che rappresenta le perdite totali di quella configurazione. Questo valore sarà in seguito utilizzato per decidere la quantità di feromone da rilasciare sul percorso della formica che ha richiesto l'analisi.

Codice di esecuzione dell'ACO

Una volta che sono state avviate tutte le istanze BFS necessarie a rispondere a tutti i profili di generazione proposti dai percorsi delle formiche, il programma è pronto per liberare gli sciami di formiche.
Come per le istanze BFS, per l'ACO vengono istanziati un numero di thread pari al numero di alberi ricoprenti moltiplicati per il numero di generatori presenti (questo perchè l'analisi deve essere eseguita dal punto di vista di tutti i nodi generatori presenti nella topologia, come spiegato nella sezione 2.2).
Il thread principale costruirà un array di threads e per ognuno di essi associerà un'istanza differente della funzione ACO_Main della classe statica `ACOControllerReboot`.
Di seguito la sezione di codice che esegue l'avvio dei thread dell'ACO.

Listing 3.15: Avvio istanze ACO

```
Thread[] threadsArrayACO = new Thread[numeroListe];
Ant[] allBestAnts = new Ant[numeroListe];
for (int i = 0; i < numeroListe; i++) {
  int localNum = i;
  log.Info("Thread ACO: " + i+"/"+this.listeDiAdiacenza.Count);
  Vector v = new DenseVector(this.nodesWithoutSlack);
  int k = 0;
  for (int j = 0; j < this.nodesTotal; j++) {
    if (j != nodo_slack) {
      v[k] = this.Scar[j];
      k++;
    }
  }
  Complex nodoSlack = this.Scar[nodo_slack];
  threadsArrayACO[i] = new Thread(() =>
    ACOControllerReboot.ACO_Main(
      this.listeDiAdiacenza[localNum],
      localNum,
      v, allBestAnts,
      nodoSlack));
}
for (int i = 0; i < numeroListe; i++)
  threadsArrayACO[i].Start();
for (int i = 0; i < numeroListe; i++)
  threadsArrayACO[i].Join();
```

Nell'ultima parte di questa sezione di codice si può notare che il thread principale si mette in attesa (Join) del completamento di tutti i thread creati.

La routine dell ACO: inizializzazione

I metodi descritti in questa sezione prendono spunto da un *Ant System* che risolve una istanza semplice di TSP, reperibile su internet [2]. Il nome della funzione principale è *ACO_Main* ed è contenuta nella classe `ACOControllerReboot`. Questa classe include anche tutti i metodi utilizzati dalla funzione sopracitata e che realizzano la ricerca della del profilo di generazione a perdite minime per mezzo dell'ACO.

ACO_Main legge dal file "App.config" i parametri elencati di seguito (molti di questi parametri sono stati descritti nel primo capitolo di questa tesi):

Listing 3.16: App.config

```
<add key="numberOfAnts" value="4"/>
<add key="numberOfIterations" value="10"/>
<add key="precisione" value="0,1"/>
<add key="alfa" value="3"/>
<add key="beta" value="2"/>
<add key="rho" value="0,01"/>
<add key="Q" value="0,000001"/>
<add key="initialPheromone" value="0,01"/>
```

Di seguito una breve spiegazione di questi parametri.

- **numberOfAnts** rappresenta il numero di formiche liberate per ogni sciame;
- **numberOfIterations** rappresenta il numero di iterazioni dell'ACO, cioè quante volte ogni sciame di formiche cercherà un nuovo percorso;
- **precisione** rappresenta la discretizzazione dei valori dei generatori;
- **alfa** rappresenta il coefficiente di influenza del feromone nella formula di calcolo di proabilità di scelta di un arco;
- **rho** rappresenta il coefficiente di decremento del feromone durante l'aggiornamento;
- **Q** rappresenta il coefficiente di incremento del feromone per un arco scelto
- **initialPheromone** rappresenta il valore iniziale di feromone presente su ogni arco.

Dopo aver letto questi valori, *ACO_Main* avvia una serie di inizializzazioni che servono alla costruzione del grafo su cui liberare le formiche per la ricerca.
L'inizializzazione permette di costruire un grafo come quello mostrato in figura 3.13:

I due cerchi estremi con bordo spesso, rappresentano, come scritto, la "tana" da cui partono le formiche e il "cibo" (o sorgente di cibo). Nel nostro caso la "tana" è rappresentata dall'avvio di una istanza ACO e il "cibo" coincide con la *query* che

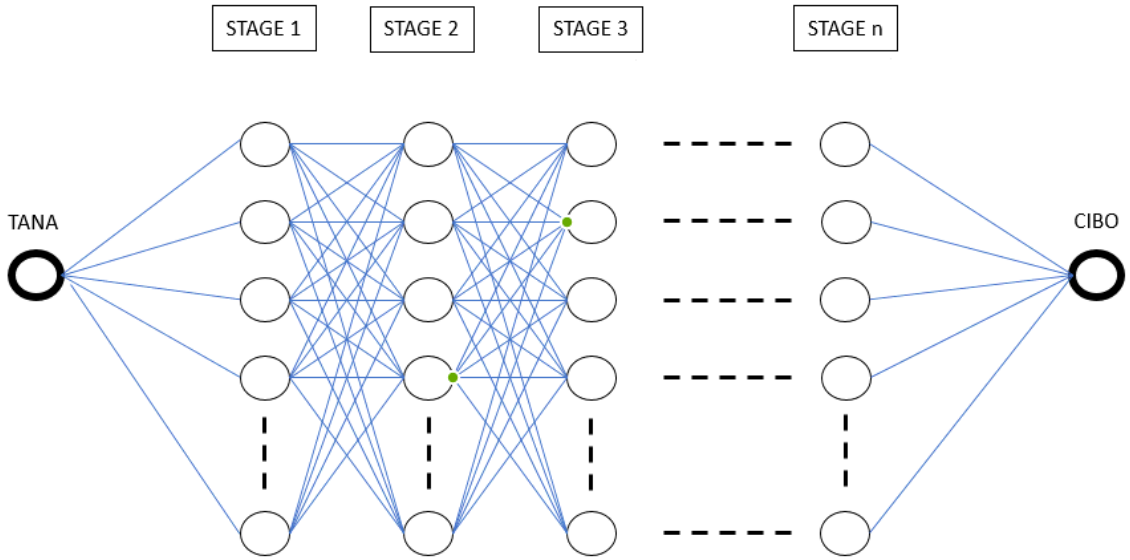

Figura 3.13: Grafo ACO

viene indirizzata all'istanza BFS associata. Ogni cerchio (a bordo sottile) all'interno del grafo, rappresenta un nodo del grafo. In questo caso i nodi possono essere definiti **stati** e ogni insieme verticale di stati può essere definito **stage**. Esistono dei vincoli di scelta degli archi all'interno del grafo: tutti i percorsi devono partire dallo stato "tana" e devono termonare sullo stato "cibo"; da ogni stato, l'arco successivo deve essere scelto nell'insieme degli stati presenti nello stage successivo. Ogni stage nel presente nel grafo, rappresenta uno dei nodi generatori nella topologia generale. Ogni stato presente uno stage rappresenta un profilo di generazione riferito al generatore associato.
Si consideri ad esempio una rete con 3 generatori, come quella presentata nella sezione 3.2.2. La combinazione fra i dati raccolti tramite il servizio REST (riferiti ai valori di generazione massima di ogni nodo generatore) ed il valore di "precisione" letto dal file `App.config` da come risultato un grafo come quello in figura 3.14.

- **valori di generazione massima (kW e kVAr):**

  `(0,4+0,09i),(0,5+0,09i),(0,2+0,09i)`

- **precisione:** 0,1

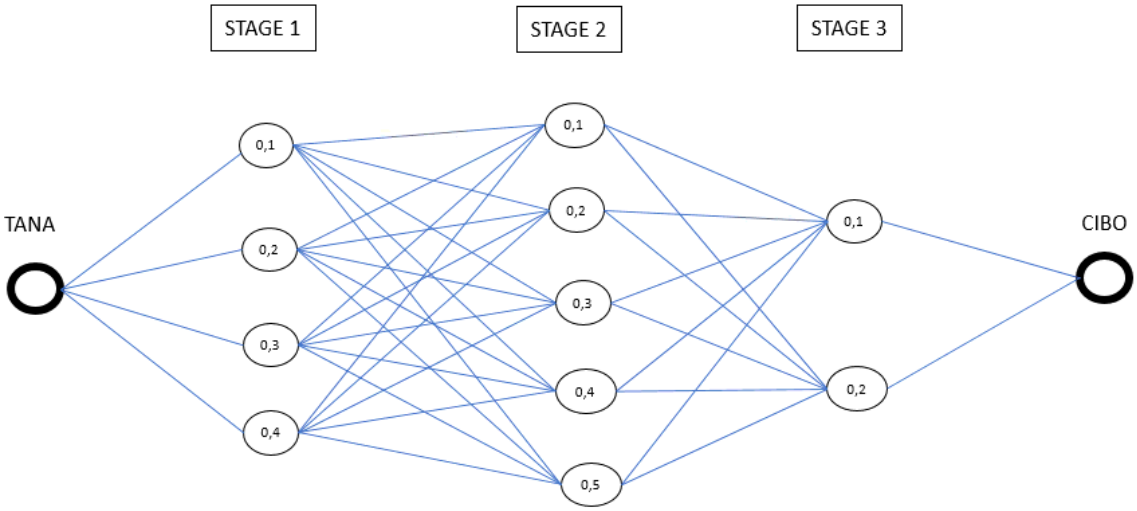

Figura 3.14: Grafo ACO 3 generatori

Dalla parte reale di ogni valore di potenza complessa, si ricavano tutti i valori dei profili di generazione, discretizzati in base alla precisione. Ogni valore ottenuto rappresenta uno stato in uno stage. Numerando gli stati, dal primo all'ultimo, iniziando da sinistra e completando uno stage alla volta si ottiene un grafo come quello mostrato in figura 3.15 e una matrice di adiacenza come quella mostrata in figura 3.16.

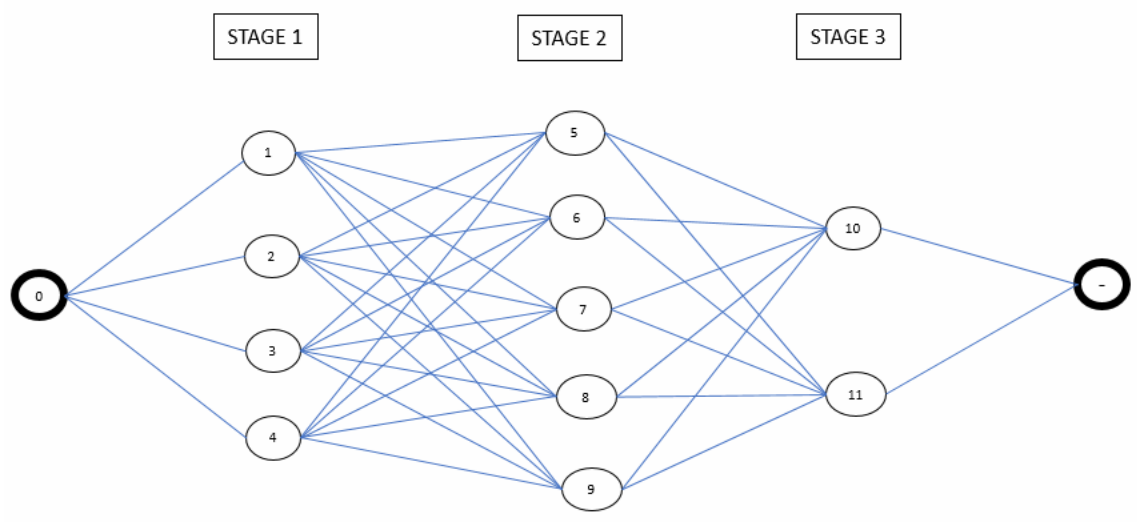

Figura 3.15: Grafo ACO numerazione stati

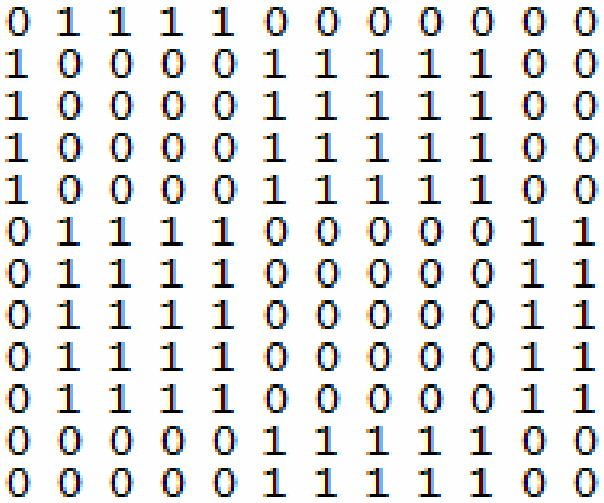

Figura 3.16: Matrice adiacenze ACO

Per eseguire questa parte di costruzione di strutture dati, sono stati sviluppati 2 metodi: *inizializzaStage(...)* che restituisce una matrice di *double* contenente per ogni generatore tutti valori discreti che può assumere; *creaMatriceDiAdiacenza(...)* che restituisce una matrice di *int* contenente le adiacenze.

Listing 3.17: inizializzaStage(...)

```
public static double[][] inizializzaStage(
                int numberOfGenerators,
                double precisione, double[] fullScale)
{
  double[][] generatorsPossibleValue =
                new double[numberOfGenerators][];
  for (int i = 0; i < numberOfGenerators; i++) {
    double prec = precisione;
    int numberOfState = (int)Math.Round(fullScale[i] / precisione);
    generatorsPossibleValue[i] = new double[numberOfState];
    for (int k = 0; k < numberOfState; k++) {
      double val = Math.Round(prec, 5);
      generatorsPossibleValue[i][k] = val;
      prec += 0.1;
    }
  }
  return generatorsPossibleValue;
}
```

Listing 3.18: creaMatriceDiAdiacenza(...)

```
private static int[][] creaMatriceDiAdiacenza(
                int numberOfStates, double[][] generatorsPossibleValue)
{
  int[][] matrice = new int[numberOfStates][];
  for (int i = 0; i <= matrice.Length - 1; i++)
    matrice[i] = new int[numberOfStates];
  int volte = 1;
  int indexGeneral = 1;
  int generatorsIndex = 0;
  for (int i = 0;
        i <= numberOfStates - 1 && indexGeneral < numberOfStates; i++)
  {
    int uni = generatorsPossibleValue[generatorsIndex].Length;
    for (int k = 0; k < volte; k++) {
    return matrice;
      for (int j = indexGeneral; j < indexGeneral + uni; j++) {
        matrice[i][j] = 1;
        matrice[j][i] = 1;
      }
    i++;
    }
    i--;
    volte = uni;
    indexGeneral += uni;
    generatorsIndex++;
  }
}
```

L'operazione successiva al caricamento dei dati che rappresentano la topologia di rete da analizzare, è l'inizializzazione delle formiche.

Listing 3.19: inizializzaFormiche(...)

```
private static int[][] inizializzaFormiche(
                double[][] generatorsPossibleValue,
                int numeroFormiche,
                int numberOfGenerators, Random random)
{
  int[][] ants = new int[numeroFormiche][];
  for (int k = 0; k <= numeroFormiche - 1; k++) {
    int start = random.Next(0, generatorsPossibleValue[0].Length);
        ants[k] =
          percorsoCasuale(generatorsPossibleValue, 0, numberOfGenerators,
              random);
  }
  return ants;
}
```

Il metodo *inizializzaFormiche(...)* calcola un percorso da "tana" a "cibo" per un numero di volte pari al *numberOfAnts* e restituisce una matrice di numeri interi su cui l'indice di riga rappresenta la formica e l'indice di colonna indica lo stage nel grafo. Il valore all'interno della matrice è invece lo stato relativo allo stage.
I percorsi associati alle formiche nel metodo *inizializzaFormiche(...)* sono totalmente casuali e sono calcolati dal metodo *percorsoCasuale(...)*.

Listing 3.20: percorsoCasuale(...)

```
private static int[] percorsoCasuale(
        double[][] generatorsPossibleValue,
        int start, int numberOfGenerators, Random random)
{
  int[] percorso = new int[numberOfGenerators];
  for (int i = 0; i <= numberOfGenerators - 1; i++) {
    percorso[i] =
        random.Next(0, generatorsPossibleValue[i].Length);
  }
  return percorso;
}
```

A questo punto rimangono ancora due inizializzazioni da operare: la matrice dei feromoni e il primo "Best Trail" (percorso migliore), sulla base delle formiche precedentemente inizializzate. La matrice dei feromoni è inizializzata come segue.

Listing 3.21: inizializzaMatriceFeromone(...)

```
private static double[][] inizializzaMatriceFeromone(
        int numberOfStates,
        int[][] matrice, double valoreFeromoneIniziale)
{
  double[][] pheromones = new double[numberOfStates][];
  for (int i = 0; i <= numberOfStates - 1; i++)
    pheromones[i] = new double[numberOfStates];
  for (int i = 0; i <= pheromones.Length - 1; i++) {
    for (int j = 0; j <= pheromones[i].Length - 1; j++) {
      if (matrice[i][j] == 1)
        pheromones[i][j] = valoreFeromoneIniziale;
    }
  }
  return pheromones;
}
```

Per trovare il primo percorso migliore, da usare succesivamente come confronto per i percorsi che verranno trovati durante il loop, si utilizza un metodo che a sua volta verrà riutilizzato all'interno della parte iterativa dell ACO:

Listing 3.22: trovaPercorsoMigliore(...)

```
private static int[] trovaPercorsoMigliore(double[][]
   generatorsPossibleValue, int[][] ants, double[] BFSResults, out double
    b, int index, int [] positions, Vector Scar, double totalLoad, double
    Abase)
{
  double bestLength = BFSQuery(
  ants[0],generatorsPossibleValue,index,
        positions,Scar,totalLoad,Abase);
  BFSResults[0] = bestLength;
  int idxBestLength = 0;
  b = bestLength;
  for (int k = 1; k <= ants.Length - 1; k++) {
    double perdite =
```

```
            BFSQuery(ants[k], generatorsPossibleValue,
                          index, positions, Scar, totalLoad,Abase);
    BFSResults[k] = perdite;
    if (perdite < bestLength) {
      bestLength = perdite;
      b = perdite;
      idxBestLength = k;
    }
  }
  int numGenerators = ants[0].Length;
  int[] bestTrail_Renamed = new int[numGenerators];
  ants[idxBestLength].CopyTo(bestTrail_Renamed, 0);
  return bestTrail_Renamed;
}
```

Nel metodo appena illustrato *trovaPercorsoMigliore(...)* vengono effettuate le query verso l'istanza BFS associata. Questa restituirà, per ogni percorso dunque per ogni formica, un valore che rappresenta le perdite in linea calcolate sulla configurazione di profili di generazione inviata dall'ACO. Per effettuare queste query viene richiamato un ulteriore funzione statica (*BFSQuery(...)*) che instaura la connessione TCP con l'istanza BFS e ne attende il risultato. Tramite la connessione TCP viene inviata, dall'ACO, una richiesta in formato di stringa che contiene un numero complesso ed un indice per ogni generatore nella rete. Di seguito un esempio di stringa di richiesta per 3 generatori:

`1:0,4+0,09i_2:0,5+0,09i_5:0,2+0,09i`

In questo caso i generatori nella topologia saranno 1,2,5.

Listing 3.23: BFSQuery(...)

```
private static double BFSQuery(
        int[] percorso, double[][] generatorsPossibleValue,
        int index, int [] positions, Vector Scar,
        double totalLoad, double Abase)
{
  string message = string.Empty;
  double coe = 0;
  double toSatisfy = Abase;
  for (int i = 0; i < percorso.Length; i++) {
    int temp = percorso[i];
```

```
    int position = positions[i];
    double immaginaryLoadValue =
          Scar[position].Imaginary;
    double realLoadValue =
          generatorsPossibleValue[i][percorso[i]];
    toSatisfy +=
          generatorsPossibleValue[i][percorso[i]];
    message = string.Format(
          message + "{0}:{1}+{2}i_",
          position, realLoadValue.ToString(),
      immaginaryLoadValue.ToString());
    message = message.Replace(',', '.');
  }
  message = message.Substring(0, message.Length - 1);
  int port = 1024 + index;
  TcpClient tcpc = new TcpClient("127.0.0.1", port);
  Byte[] data = System.Text.Encoding.UTF8.GetBytes(message);
  NetworkStream stream = tcpc.GetStream();
  stream.Write(data, 0, data.Length);
  string received_double = string.Empty;
  int j;
  while ((j = stream.Read(data, 0, data.Length)) != 0) {
    received_double =
      System.Text.Encoding.UTF8.GetString(data, 0, j);
    break;
  }
  int bytes = stream.Read(data, 0, data.Length);
  string responseData =
    System.Text.Encoding.UTF8.GetString(data, 0, bytes);
  coe = double.Parse(received_double);
  stream.Close();
  if (toSatisfy < totalLoad)
    return double.MaxValue;
  return coe;
}
```

La routine dell ACO: loop

Come spiegato nel primo capitolo, all'interno del loop, una volta "liberate" le formiche, le due operazioni principali sono:

- **Costruzione dei percorsi per ogni formica**

  Sulla base dei percorsi disponibili.

- **Aggiornamento del feromone**

  Sulla base dei risultati ottenuti dal BFS.

Oltre alle due operazioni elencate, dopo ogni iterazione, viene salvata la "best ant", che nel nostro caso è rappresentata dalla formica che ha ricevuto la quantità di perdite minore dal BFS.
Il numero di iterazioni del ciclo è deciso a priori e dipende dalla velocità di convergenza dell'algoritmo. Questo numero cresce al crescere della precisione impostata (più si rende piccolo il valore di discretizzazione dei profili di generazione e più sarà grande il numero di stati per ogni stage) ed al crescere della topologia di rete stessa (in realtà se il numero di generatori nella rete rimane costante anche il numero di iterazioni utili può restare uguale).

Listing 3.24: ACO Loop

```
while (counter < numeroDiIterazioni) {
  nuoviPercorsiPerFormiche(generatorsPossibleValue,
  numberOfGenerators, ants, pheromones, matriceAdiacenza,
  random, alpha, beta);
  UpdatePheromones(generatorsPossibleValue,
  numberOfGenerators, pheromones, ants, matriceAdiacenza,
  BFSResults, statesStages, rho, Q);
  double currentBestBFSResult = 0;
  int[] currBestTrail =
  trovaPercorsoMigliore(generatorsPossibleValue,
  ants, BFSResults, out currentBestBFSResult, index,
  generatorPosition,v,totalLoad,Abase);
  if (currentBestBFSResult < bestBFSResult) {
    bestBFSResult = currentBestBFSResult;
    percorsoMigliore = currBestTrail; }
  counter += 1; }
```

All'interno del loop, vengono quindi invocati i due metodi *nuoviPercorsiPerFormiche(...)* e *UpdatePheromones(...)*.
Il primo fa uso di un ulteriore metodo *trovaPercorso(...)* per riempire, con nuovi valori, la matrice delle formiche sopra descritta (contenente un percorso per ogni formica). I percorsi vengono selezionati calcolando la probabilità di scelta di ogni arco come descritto nel capitolo 1 (nel caso specifico di questa tesi le probabilità dipendono soltanto dal valore di feromone associato all'arco e dal coefficiente di influenza α).
Il secondo invece, scorre la matrice dei feromoni e incrementa le entry corrispondenti ai percorsi delle formiche di una quantità che dipende dal coefficiente Q e dal risultato della query verso il BFS (perdite totali in linea). Su tutte le entry della matrice dei feromoni che non appartengono a nessun percorso scelto dalle formiche viene effettuato un decremeto di feromone che dipende dal coefficiente ρ. Di seguito tutti i metodi utilizzati per effettuare le opearazioni descritte.

Listing 3.25: nuoviPercorsiPerFormiche(...)

```
private static void nuoviPercorsiPerFormiche(
        double[][] generatorsPossibleValue,
        int numberOfGenerators, int[][] ants,
        double[][] pheromones, int[][] matrice,
        Random random, double alpha, double beta)
{
  int numStates = pheromones.Length;
  for (int k = 0; k <= ants.Length - 1; k++) {
    int start =
       random.Next(0, generatorsPossibleValue[0].Length);
    int[] nuovoPercorso =
       trovaPercorso(generatorsPossibleValue,
       numberOfGenerators, k, start, pheromones,
       matrice, random, alpha, beta);
  ants[k] = nuovoPercorso;
  }
}
```

Listing 3.26: trovaPercorso(...)

```
private static int[] trovaPercorso(
        double[][] generatorsPossibleValue,
        int numberOfGenerators, int k, int start,
        double[][] pheromones, int[][] matrice,
        Random random, double alpha, double beta)
{
  int lunghezzaPercorso = numberOfGenerators;
  int[] percorso = new int[lunghezzaPercorso];
  percorso[0] = start;
  int indiceDiRiga = start + 1;
  int next = -1;
  for (int i = 0; i <= lunghezzaPercorso - 2; i++) {
    if (i == 0) {
      next = trovaStatoSuccessivo(
          generatorsPossibleValue, k,
          pheromones, matrice, indiceDiRiga,
          i + 1, random,alpha, beta);
    }
    else {
      indiceDiRiga =
         next + generatorsPossibleValue[i - 1].Length + 1;
      next = trovaStatoSuccessivo(
      generatorsPossibleValue, k,
      pheromones, matrice, indiceDiRiga,
      i + 1, random,alpha,beta);
    }
    percorso[i + 1] = next;
  }
  return percorso;
}
```

Listing 3.27: trovaStatoSuccessivo(...)

```
private static int trovaStatoSuccessivo(
        double[][] generatorsPossibleValue,
        int k, double[][] pheromones, int[][] matrice,
        int state, int stage, Random random,
        double alpha, double beta)
{
  double[] probs = calcolaProbabilita(
      generatorsPossibleValue, k, pheromones,
      matrice, state, stage,alpha,beta);
  double[] cumul = new double[probs.Length + 1];
  for (int i = 0; i <= probs.Length - 1; i++)
    cumul[i + 1] = cumul[i] + probs[i];
  double p = random.NextDouble();
  for (int i = 0; i <= cumul.Length - 2; i++) {
    if (p >= cumul[i] && p < cumul[i + 1]) {
      return i;
    }
  }
}
```

Listing 3.28: calcolaProbabilita(...)

```
private static double[] calcolaProbabilita(
        double[][] generatorsPossibleValue, int k,
        double[][] pheromones, int[][] matrice,
        int state, int stage, double alpha, double beta)
{
  int reachableStates =
        generatorsPossibleValue[stage].Length;
  double[] taueta = new double[reachableStates];
  double sum = 0.0;
  int firstIndexOfPheromone = state;
  bool trovato = false;
  for (int i = state; i < pheromones.Length && !trovato; i++) {
    if (pheromones[state][i] > 0) {
      trovato = true;
      firstIndexOfPheromone = i;
    }
  }
```

```
  int indexOnPheromone = firstIndexOfPheromone;
  for (int i = 0; i <= taueta.Length - 1; i++) {
    taueta[i] =
       Math.Pow(pheromones[state][indexOnPheromone], alpha) *
         Math.Pow((1.0 / 1), beta);
    if (taueta[i] < 0.0001) {
      taueta[i] = 0.0001;
    }
    else if (taueta[i] >
          (double.MaxValue / (reachableStates * 100))) {
       taueta[i] = double.MaxValue / (reachableStates * 100);
    }
    indexOnPheromone++;
    sum += taueta[i];
  }
  double[] probs = new double[reachableStates];
  for (int i = 0; i <= probs.Length - 1; i++) {
    probs[i] = taueta[i] / sum;
  }
return probs;
}
```

Listing 3.29: UpdatePheromones(...)

```
private static void UpdatePheromones(
        double[][] generatorsPossibleValue,
        int numberOfGenerators, double[][] pheromones,
        int[][] ants, int[][] matrice,
        double[] BFSResults, int[] statesStages,
        double rho, double Q)
{
  for (int i = 0; i <= pheromones.Length - 1; i++) {
    for (int j = i + 1; j <= pheromones[i].Length - 1; j++) {
      if (matrice[i][j] == 0)
        continue;
      for (int k = 0; k <= ants.Length - 1; k++) {
        double BFSresultAntK = BFSResults[k];
        double decrease = (1.0 - rho) * pheromones[i][j];
        double increase = 0.0;
        if (trovaArcoInPercorso(generatorsPossibleValue,
```

```
            numberOfGenerators, i, j,
            ants[k], statesStages) == true)
        {
            increase = (Q / BFSresultAntK);
        }
        pheromones[i][j] = decrease + increase;
        if (pheromones[i][j] < 0.0001) {
          pheromones[i][j] = 0.0001;
        }
        else if (pheromones[i][j] > 100000.0) {
          pheromones[i][j] = 100000.0;
        }
        pheromones[j][i] = pheromones[i][j];
      }
    }
  }
}
```

Listing 3.30: trovaArcoInPercorso(...)

```
private static bool trovaArcoInPercorso(
        double[][] generatorsPossibleValue,
        int numberOfGenerators, int stateX, int stateY,
        int[] percorso, int[] statesStages)
{
  bool inTrail = false;
  int[] vettoreAppoggio = new int[statesStages.Length];
  vettoreAppoggio[0] = -1; vettoreAppoggio[1] = 0;
  int value = statesStages[1];
  int k = 1;
  for (int i = 2; i < statesStages.Length; i++) {
    if (statesStages[i] == value) {
      vettoreAppoggio[i] = k;
      k++;
    } else {
      vettoreAppoggio[i] = 0;
      value = statesStages[i];
      k = 1;
    }
  }
```

```
int precNode = 0;
int[][] archiDiQuestaTrail = new int[percorso.Length][];
int searchingFor = 0;
int indexone = 1;
for (int i = 0; i < numberOfGenerators; i++) {
  archiDiQuestaTrail[i] = new int[2];
  if (i == 0) {
    archiDiQuestaTrail[i][0] = precNode;
    for (int j = indexone;
      j < generatorsPossibleValue[i].Length; j++)
    {
      if (statesStages[j] == searchingFor) {
        if (percorso[i] == vettoreAppoggio[j]) {
          archiDiQuestaTrail[i][1] = j;
          precNode = j;
          searchingFor++;
          break;
        } } }
    indexone += generatorsPossibleValue[i].Length;
  } else {
    archiDiQuestaTrail[i][0] = precNode;
    for (int j = indexone;
      j < indexone + generatorsPossibleValue[i].Length; j++)
    {
       if (statesStages[j] == searchingFor) {
         if (percorso[i] == vettoreAppoggio[j]) {
           archiDiQuestaTrail[i][1] = j;
           precNode = j;
           searchingFor++;
           break;
         } } }
    indexone += generatorsPossibleValue[i].Length;
  } }
for (int i = 0; i < archiDiQuestaTrail.Length; i++) {
  if (stateX == archiDiQuestaTrail[i][0] &&
    stateY == archiDiQuestaTrail[i][1])
       inTrail = true;
}
return inTrail; }
```

Terminazione dei Thread e valutazione dei risultati

L'ultima parte dell'algoritmo riguarda la scelta del percorso a minori perdite in linea. Ogni thread istanziato per l'esecuzione dell'ACO, aggiunge un valore sul vettore *allBestAnts*, in cui sono contenuti degli oggetti `Ant`. In ognuno di questi oggetti sono memorizzati: percorso della formica che ha ottenuto il risultato di perdite minime nel suo thread (un array di *int* contenente gli indici degli stati per ogni stage attraversato); il valore di perdite minime trovato; i valori da mandare ai generatori come profili di generazione. Da queste "Best Ants" viene selezionata quella con le perdite minime, e vengono registrati sul Database i valori di attuazione per i generatori.

Listing 3.31: Valutazione della Best Ant of Ever

```
double best = double.MaxValue;
for (int i = 0; i < allBestAnts.Length; i++) {
  double d = allBestAnts[i].getBFSResultCoef();
  if (d < best) {
    spanningTreeIndex = i;
    best = d;
  }
}
Ant bestOfBests = allBestAnts[spanningTreeIndex];
string obtained = "";
double [] res = bestOfBests.getCoefs();
double [] resWithSlack = new double [res.Length+1];
resWithSlack[this.selectedTopology] = 1000;
int kk = 0;
for (int i = 0; i < resWithSlack.Length; i++) {
  if (resWithSlack[i] == 0) {
    resWithSlack[i] = res[kk];
    kk++;
  }
}
for (int i = 0; i < resWithSlack.Length; i++) {
  obtained +=
        ((resWithSlack[i] / precisione)).ToString() + "; ";
}
this.acodbc.UpdatePowers(obtained);
```

A fine elaborazione il Form sopra descritto ci apparirà come in figura 3.17.

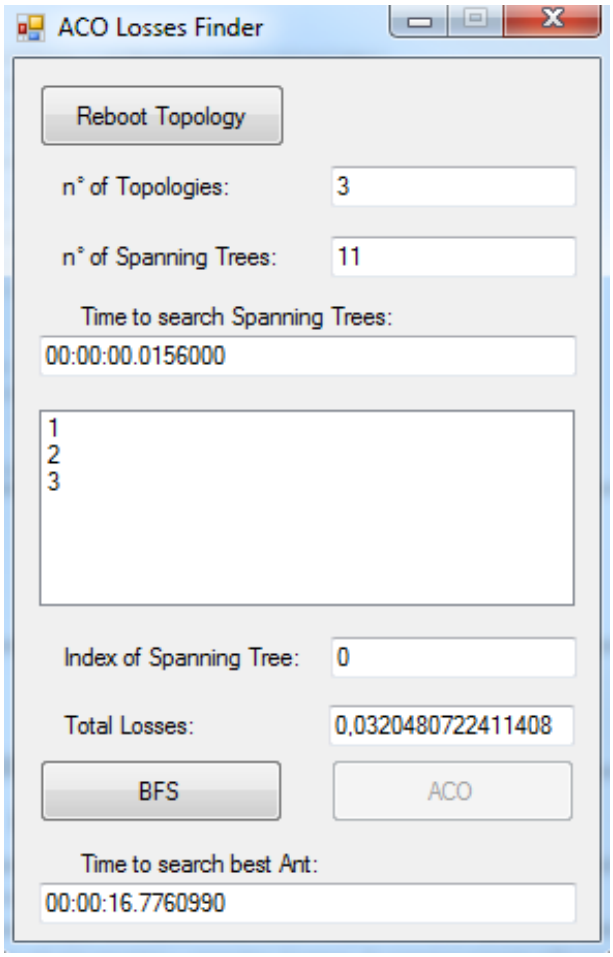

Figura 3.17: Form Finale

Sarà indicato: quale degli alberi ricoprenti è stato scelto, valore minimo di perdite trovate e tempo di esecuzione.

3.2.4 Database

Il database utilizzato è un "MySql" ed è stato creato e configurato tramite "phpmyadmin" ed eseguito nel contesto del client XAMPP.
Il database "aco_db" si compone di due tabelle semplici: la tabella "generators" viene aggiornata ad ogni esecuzione e memorizza il numero di generatori presenti nella topologia e un timestamp di aggiornamento; la tabella "potenzedaattuare" viene aggiornata ogni volta che l'ACO e il BFS terminano un esecuzione e contiene i valori da mostrare

sul Dashboard.

Di seguito le due tabelle su "phpmyadmin"

#	Nome	Tipo	Codifica caratteri	Attributi	Null	Predefinito	Extra
1	id	int(11)			No	*Nessuno*	AUTO_INCREMENT
2	numberOfGenerator	int(11)			No	*Nessuno*	
3	timestamp	varchar(30)	latin1_swedish_ci		No	*Nessuno*	

Figura 3.18: Tabella generators

#	Nome	Tipo	Codifica caratteri	Attributi	Null	Predefinito	Extra
1	id	int(11)			No	*Nessuno*	AUTO_INCREMENT
2	potenze	text	latin1_swedish_ci		No	*Nessuno*	
3	timestamp	datetime			No	*Nessuno*	

Figura 3.19: Tabella potenzedaattuare

Per l'accesso ai dati nel programma "ACO Losses Finder" è presente la classe "ACODBConnect" che implementa il metodo UPDATE di SQL.

Listing 3.32: ACODBConnect

```
public void UpdateNumberOfGenerators(Vector v)
{
  int numberOfGenerator = 0;
  for (int i = 0; i < v.Count; i++) {
        if (v[i].Real < 0)
        numberOfGenerator++;
  }
  DateTime dt = DateTime.Now;
  string timestamp = dt.ToString();
  string query = "UPDATE generators SET numberOfGenerator=" +
     numberOfGenerator + ", timestamp='" + timestamp + "' WHERE id=1";
  if (this.OpenConnection() == true) {
        MySqlCommand cmd = new MySqlCommand();
        cmd.CommandText = query;
        cmd.Connection = connection;
        cmd.ExecuteNonQuery();
        this.CloseConnection();
  } }
```

3.2.5 Dashboard

La Dashboard è una semplice Web Application creata in Visual Studio, che utilizza una libreria JavaScript (Drink.js). In figura un esempio di interfaccia web.

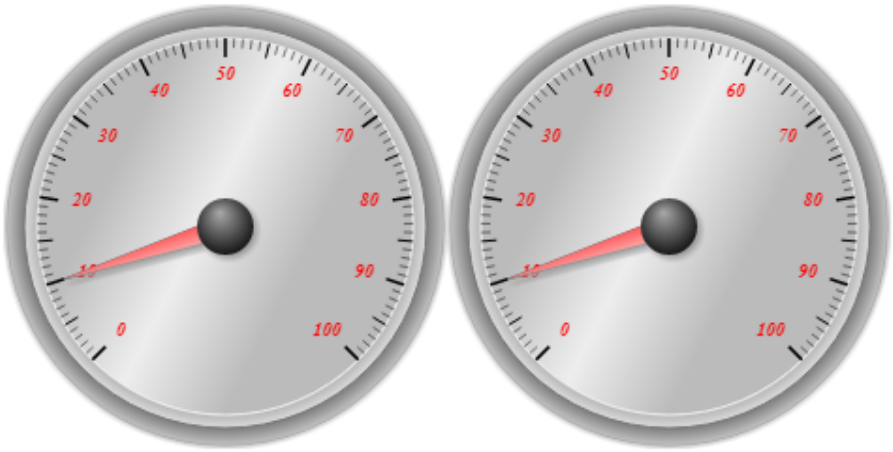

Figura 3.20: Interfaccia dashboard

La Web Application utilizza un "behind code" collegato ad una semplicissima pagina web; il "behind code" istanzia una classe "Dashboard" che estende la classe "System.Web.UI.Page" del Framework reagendo al caricamento della pagina web associata. Essendo necessario l'accesso ai dati del database, è stata creata una classe di supporto per l'accesso ai dati, che oltre a gestire apertura e chiusura della connessione con il database, espone due metodi per l'accesso alle due tabelle. Questa classe è stata scritta utilizzando un templeate di accesso ai dati online [7].

Listing 3.33: ACODBConnectDashboard

```
public class ACODBConnectDashboard
{
private MySqlConnection connection;
private string server;
private string port;
private string database;
private string uid;
private string password;
public ACODBConnectDashboard() {
private void Initialize() { ... }
private bool OpenConnection() { ... }
```

```
private bool CloseConnection() { ... }
public int SelectNumberOfGenerators() {
  int numberOfGenerators = 0;
  string query = "SELECT numberOfGenerator FROM generators";
  if (this.OpenConnection() == true) {
        MySqlCommand cmd = new MySqlCommand(query, connection);
        MySqlDataReader dataReader = cmd.ExecuteReader();
        while (dataReader.Read()) {
          Object[] values = new Object[dataReader.FieldCount];
          int fieldCount = dataReader.GetValues(values);
          numberOfGenerators = (int)values[0];
        }
        dataReader.Close();
        this.CloseConnection();
        return numberOfGenerators;
  }
  else
        return numberOfGenerators;
}

public string SelectPotenze()
{
  string query = "SELECT potenze FROM potenzedaattuare";
  string list = "";
  if (this.OpenConnection() == true) {
        MySqlCommand cmd = new MySqlCommand(query, connection);
        MySqlDataReader dataReader = cmd.ExecuteReader();
        while (dataReader.Read()) {
          Object[] values = new Object[dataReader.FieldCount];
          int fieldCount = dataReader.GetValues(values);
          list = (string)values[0];
        }
        dataReader.Close();
        this.CloseConnection();
        return list;
  }
  else
        return list;
 } }
```

Listing 3.34: Pagina Web Dasboard

```
<%@ Page Language="C#"
        AutoEventWireup="true"
        CodeBehind="Dashboard.aspx.cs"
        Inherits="ACOGeneratorsDashboard.Dashboard" %>

<!DOCTYPE html>
<html xmlns="http://www.w3.org/1999/xhtml">
  <head runat="server">
    <meta http-equiv="Content-Type" content="text/html; charset=utf-8"/>
    <title>Generator Dashboard</title>
    <script type="text/javascript" src="Scripts/Drinks.js"></script>
    <link rel="stylesheet" type="text/css" href="Css/style.css">
    <meta http-equiv="refresh" content="10"/>
  </head>
  <body>
    <div id="container">
     <div id="inner" runat="server">
        <!-- Drinks.js add component here -->
     </div>
    </div>
  </body>
</html>
```

Listing 3.35: Code Behind Dasboard

```
public partial class Dashboard : System.Web.UI.Page
{
  public ACODBConnectDashboard acodbc;
  int numberOfGenerators = 0;
  string powers;
  int[] potenze;
  public Dashboard()
  {
        this.acodbc = new ACODBConnectDashboard();
        this.numberOfGenerators = this.acodbc.SelectNumberOfGenerators();
        this.powers = this.acodbc.SelectPotenze();
        this.potenze = new int[this.numberOfGenerators];
        string [] potenze_string = this.powers.Split(';');
        for (int i = 0; i < this.numberOfGenerators; i++)
```

```
        {
          this.potenze[i] = int.Parse(potenze_string[i]);
        }
  }

  protected void Page_Load(object sender, EventArgs e)
  {
        for (int i = 0; i < this.numberOfGenerators; i++)
        inner.InnerHtml += "<div class=\"child\"> <display value="
        + this.potenze[i] + " id=\"gau"
        + i + "\" refresh=\"5\"></display> </div>";
  }
}
```

CAPITOLO 4

Risultati

Per comprendere quanto questo progetto possa essere utile al fine di ottenere un risparmio energetico, sono stati effettuati molti test. Sono state proposte all'algoritmo topologie di dimensioni diverse, con diversa magliatura e diversi valori di carico e generazione disponibili. Sono state proposte topologie di dimensioni uguali ma con dati sui rami (resistenze e reattanze) variabili. Infine i test sono stati ripetuti variando il numero di formiche ed il numero di iterazioni. Verrà utilizzata una rete di riferimento per mostrare i risultati più evidenti; la topologia di questa rete è molto semplice e debolmente magliata, in quanto la prima considerazione da mettere alla luce riguarda la crescita esponenziale del tempo di costruzione delle strutture dati all'aumentare del numero di archi. Il programma esegue l'analisi su una fotografia della rete che include carichi e generatori in un dato istante. Prima di iniziare l'analisi però vengono generate tante topologie quanti sono gli alberi ricoprenti della rete base. Per comprendere l'aumento esponenziale di seguito un esempio.
Si consideri una rete a 10 nodi e 15 archi. Il grafo che reappresenta questa rete è composta quindi da 10 nodi; un qualsiasi albero ricoprente di questo grafo conterrà esattamente *(N-1)* archi con N numero di nodi. La funzione *Combinations(...)*, descritta nei capitoli precedenti, genera tutte le combinazioni senza ripetizione nell'insieme dei 15 archi (n), estraendone quindi 9 (k)alla volta. Utilizzando la formula combinatoria

$$C_{n,k} = \frac{n!}{k!(n-k)!} \tag{4.1}$$

si ottengono, nel caso specifico, 5005 combinazioni. Se si aggiunge un solo arco, il risultato della stessa operazione genera 11440 combinazioni. Solo una parte di questi

sarà un albero ricoprente, ma il programma deve analizzarle tutte per scoprirlo, di conseguenza il tempo di costruzione cresce esponenzialmente.
Di conseguenza a questa prima considerazione, come detto precedentemente, le reti utilizzate come test, risultano debolmente magliate (Es: 6 Nodi, 7-8 Archi; 10 Nodi, 10-11 Archi; 13 Nodi, 13-14 Archi). La rete in figura rappresenta una delle topologie testate con 13 nodi e 13 archi e produce 6 alberi ricoprenti.

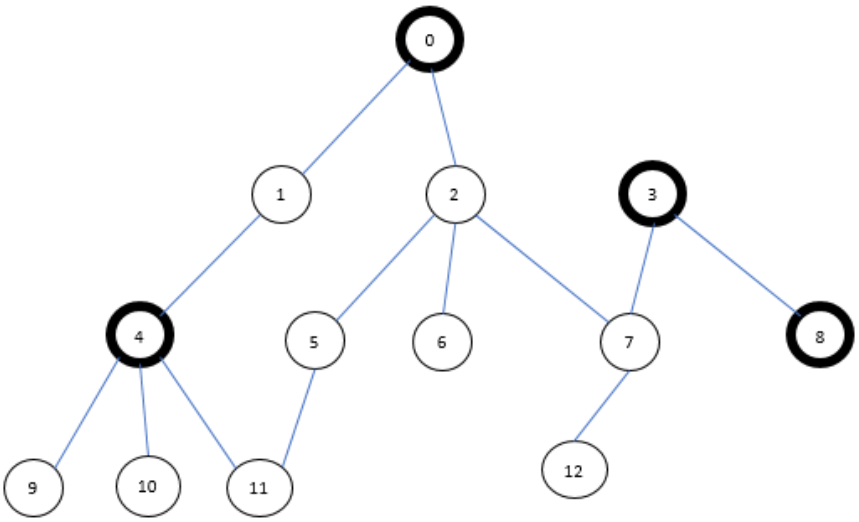

Figura 4.1: Topologia Test

Nell'illustrzione i nodi con bordo spesso rappresentano i generatori in rete. Ecco i dati di configurazione utilizzati mostrati in figura.

4	13											
0	3	4	8									
0	1	1	0	0	0	0	0	0	0	0	0	0
1	0	0	0	1	0	0	0	0	0	0	0	0
1	0	0	0	0	1	1	1	0	0	0	0	0
0	0	0	0	0	0	0	1	1	0	0	0	0
0	1	0	0	0	0	0	0	0	1	1	1	0
0	0	1	0	0	0	0	0	0	0	0	1	0
0	0	1	0	0	0	0	0	0	0	0	0	0
0	0	1	1	0	0	0	0	0	0	0	0	1
0	0	0	1	0	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	1	0	0	0	0	0	0	0	0
0	0	0	0	1	1	0	0	0	0	0	0	0
0	0	0	0	0	0	0	1	0	0	0	0	0

Figura 4.2: Matrice Topologia Test

	13	0_1	0_2	1_4	2_5	2_6	2_7	3_7	3_8	4_9	4_10	4_11	5_11	7_12
R		0.007820	0.007820	0.011730	0.011730	0.015640	0.015640	0.015640	0.011730	0.007820	0.015640	0.015640	0.015640	0.011730
X		0.002120	0.002120	0.003180	0.003180	0.004240	0.004240	0.004240	0.003180	0.002120	0.004240	0.004240	0.004240	0.003180
	13	3_1	3_2	1_4	2_5	2_6	2_7	0_7	0_8	4_9	4_10	4_11	5_11	7_12
R		0.007820	0.007820	0.011730	0.011730	0.015640	0.015640	0.015640	0.011730	0.007820	0.015640	0.015640	0.015640	0.011730
X		0.002120	0.002120	0.003180	0.003180	0.004240	0.004240	0.004240	0.003180	0.002120	0.004240	0.004240	0.004240	0.003180
	13	4_1	4_2	1_0	2_5	2_6	2_7	3_7	3_8	0_9	0_10	0_11	5_11	7_12
R		0.007820	0.007820	0.011730	0.011730	0.015640	0.015640	0.015640	0.011730	0.007820	0.015640	0.015640	0.015640	0.011730
X		0.002120	0.002120	0.003180	0.003180	0.004240	0.004240	0.004240	0.003180	0.002120	0.004240	0.004240	0.004240	0.003180
	13	8_1	8_2	1_4	2_5	2_6	2_7	3_7	3_0	4_9	4_10	4_11	5_11	7_12
R		0.007820	0.007820	0.011730	0.011730	0.015640	0.015640	0.015640	0.011730	0.007820	0.015640	0.015640	0.015640	0.011730
X		0.002120	0.002120	0.003180	0.003180	0.004240	0.004240	0.004240	0.003180	0.002120	0.004240	0.004240	0.004240	0.003180

Figura 4.3: Resistenze e Reattanze

0	-1,9	0,04	-1,9	0,04	-1,9	0,04	-1	0,04	-1	0,04	-1,9	0,1
1	0,1	0,04	0,2	0,04	0,1	0,04	0,1	0,04	0,1	0,04	0,1	0,04
2	0,1	0,03	0,2	0,03	0,3	0,03	0,3	0,03	0,3	0,03	0,3	0,03
3	-1,9	0,03	-1,9	0,03	-1,9	0,03	-2,8	0,03	-0,9	0,03	-1,9	0,1
4	-1,9	0,06	-1,9	0,06	-1,9	0,06	-1,9	0,06	-2,8	0,06	-1,9	0,1
5	0,1	0,04	0,2	0,04	0,1	0,04	0,1	0,04	0,1	0,04	0,1	0,04
6	0,1	0,04	0,2	0,04	0,3	0,04	0,3	0,04	0,3	0,04	0,3	0,04
7	0,1	0,03	0,2	0,03	0,1	0,03	0,1	0,03	0,1	0,03	0,1	0,03
8	-1,9	0,06	-1,9	0,06	-1,9	0,06	-1,9	0,06	-1,9	0,06	-1,9	0,06
9	0,1	0,06	0,2	0,06	0,1	0,06	0,1	0,06	0,1	0,06	0,1	0,06
10	0,1	0,04	0,2	0,04	0,3	0,04	0,3	0,04	0,3	0,04	0,3	0,04
11	0,1	0,04	0,2	0,04	0,1	0,04	0,1	0,04	0,1	0,04	0,1	0,04
12	0,1	0,03	0,2	0,03	0,3	0,03	0,3	0,03	0,3	0,03	0,3	0,03
	1		2		3		4		5		6	

Figura 4.4: Carichi e Generatori Test

Nella figura 4.4 vi sono alcuni "test case" eseguti sulla medesima rete e rappresentano: (1) basso carico, (2) carico medio, (3) carico misto, (4)(5) carico misto e generazione mista, (6) carico misto e potenza reattiva media dei generatori.
Tutte le configurazioni sono state testate con un numero di formiche e iterazioni differenti (illustrate: formiche 6 iterazioni 15; formiche 20, iterazioni 20). Per semplicità verranno illustrate solo le perdite in termini di potenza attiva (parte Reale).
In figura 4.5 possiamo vedere i risultati. A sinistra la serie di test, sulle istanze proposte, eseguite con 6 formiche su 15 iterazioni; a destra con 20 formiche su 20 iterazioni.

0	ST	Perdite	Profili di Generazione
0	1	0,015443708	100; 40; 20; 30;
3	0	0,012017131	40; 100; 20; 30;
4	0	0,007576836	40; 20; 100; 30;
8	1	0,013746317	40; 20; 30; 100;
1	ST	Perdite	Profili di Generazione
0	2	0,017689141	100; 10; 140; 30;
3	1	0,015724473	50; 20; 100; 50;
4	1	0,016247941	50; 20; 100; 50;
8	1	0,020771741	30; 50; 80; 100;
2	ST	Perdite	Profili di Generazione
0	1	0,019853878	100; 40; 20; 30;
3	2	0,024519832	110; 100; 40; 40;
4	1	0,015602575	30; 30; 100; 50;
8	1	0,022077506	50; 20; 50; 100;
3	ST	Perdite	Profili di Generazione
0	2	0,024554675	100; 10; 140; 30;
3	2	0,023413111	90; 100; 50; 40;
4	1	0,015602579	30; 30; 100; 50;
8	1	0,0222788	30; 30; 50; 100;
4	ST	Perdite	Profili di Generazione
0	0	0,016802497	100; 20; 20; 30;
3	1	0,024005484	60; 100; 50; 40;
4	1	0,017717594	50; 20; 100; 50;
8	0	0,022374013	10; 70; 40; 100;
5	ST	Perdite	Profili di Generazione
0	1	0,021243902	100; 40; 20; 30;
3	4	0,02863752	50; 100; 60; 50;
4	1	0,017075683	20; 40; 100; 60;
8	1	0,031478994	20; 40; 60; 100;

	ST	Perdite	Profili di Generazione
0	0	0,005001437	100; 20; 40; 10;
3	0	0,011078136	20; 100; 40; 10;
4	1	0,005906047	20; 10; 100; 20;
8	0	0,012373782	20; 40; 10; 100;
	ST	Perdite	Profili di Generazione
0	0	0,012713375	100; 20; 40; 10;
3	4	0,020119649	50; 100; 80; 10;
4	1	0,014977724	40; 30; 100; 40;
8	0	0,019354155	30; 70; 40; 100;
	ST	Perdite	Profili di Generazione
0	0	0,013740577	100; 20; 40; 10;
3	4	0,021158736	50; 100; 80; 10;
4	1	0,015608945	20; 40; 100; 60;
8	1	0,022024862	20; 40; 60; 100;
	ST	Perdite	Profili di Generazione
0	0	0,013740576	100; 20; 40; 10;
3	4	0,020718431	50; 100; 70; 10;
4	1	0,015828075	10; 60; 100; 60;
8	0	0,02275231	50; 40; 20; 100;
	ST	Perdite	Profili di Generazione
0	2	0,013267745	100; 30; 80; 10;
3	4	0,020804888	70; 100; 60; 20;
4	1	0,015142386	20; 60; 100; 50;
8	1	0,022077505	50; 20; 50; 100;
	ST	Perdite	Profili di Generazione
0	0	0,015153516	100; 20; 40; 10;
3	4	0,025185913	50; 100; 80; 10;
4	1	0,017075683	20; 40; 100; 60;
8	1	0,031478994	20; 40; 60; 100;

Figura 4.5: Risultati

In figura 4.5 il primo indice indica l'istanza test eseguita, l'acronimo ST indica l'indice dell'albero ricoprente selezionato come migliore fra i disponibili (l'albero su cui è stata calcolata la perdita minima). I valori in tabella indicano, per ogni riga, l'esecuzione dell'algoritmo impostando uno dei generatori che nella topologia corrispondono all'indice di sinistra.
I tempi di calcolo per ogni esecuzione (in questo caso per ogni riga) sono in media 9,5 secondi per la parte sinistra dei risultati (con un minor numero di formiche) e 34,5 per la parte destra (con un maggior numero di formiche). Questo dato per far notare che il tempo di esecuzione cresce linearmente al crescere del numero di formiche e iterazioni. In alcuni casi però risulta necessario utilizzare un maggior numero di formiche. La quantità di formiche da utilizzare, in questo progetto dipende dal numero di generatori presenti in rete de dal valore di precisione (che indica il numero di stati per ogni stage). Non dimensionando correttamente il quantitativo di formiche da liberare si possono trovare dei sub-ottimi come si può notare dalla prima riga dei risultati.

0	ST	Perdite	Profili di Generazione
0	1	0,015443708	100; 40; 20; 30;

	ST	Perdite	Profili di Generazione
0	0	0,005001437	100; 20; 40; 10;

Figura 4.6: Dimensionamento Formiche

Nella prima esecuzione, sulla base dei parametri impostati, non tutti i percorsi possibili sono stati selezionati dalle formiche disponibili. Infatti nell'esecuzione di destras è stata trovata una configurazione di generazione che utilizza un albero ricoprente differente e che fa diminuire di un fattore 10 le perdite.
Di seguito, i dati e i risultati di altre esecuzioni su una topologia a 6 nodi con 2 generatori. In questo caso, date le minori dimensioni della topologia, il numero di formiche presenti non cambia il risultato. L'algoritmo in questo caso converge velocemente con l'utilizzo di 5 formiche su 14 iterazioni.

2	6					
0	5					
0	1	1	0	0	0	
1	0	1	1	0	0	
1	1	0	0	0	1	
0	1	0	0	1	1	
0	0	0	1	0	0	
0	0	1	1	0	0	

Figura 4.7: Topologia: 6 Nodi 2 Generatori

7	0_1	0_2	1_2	1_3	2_5	3_4	3_5
R	0.007820	0.015640	0.011730	0.007820	0.015640	0.011730	0.015640
X	0.002120	0.004240	0.003180	0.002120	0.004240	0.003180	0.004240
7	5_1	5_2	1_2	1_3	2_0	3_4	3_0
R	0.007820	0.015640	0.011730	0.007820	0.015640	0.011730	0.015640
X	0.002120	0.004240	0.003180	0.002120	0.004240	0.003180	0.004240

Figura 4.8: Dati sui rami; 6 Nodi 2 Generatori

		tempo exe	00:00:09.8995662
	ST	Perdite	Time
0	2	0,008409602	100; 70;
5	2	0,0127405	70; 100;
	ST	Perdite	Time
0	0	0,04279209	100; 30;
5	3	0,05451168	40; 100;
	ST	Perdite	Time
0	1	0,13128054	100; 120;
5	2	0,12705092	270; 100;
	ST	Perdite	Time
0	1	0,0672698	100; 70;
5	0	0,06949305	120; 100;
	ST	Perdite	Time
0	6	0,12512734	100; 220;
5	3	0,12142813	90; 100;
	ST	Perdite	Time
0	1	0,09638844	100; 130;
5	2	0,09393176	190; 100;
	ST	Perdite	Time
0	1	0,09461267	100; 110;
5	2	0,09439866	210; 100;
	ST	Perdite	Time
0	1	0,03113999	100; 70;
5	5	0,012937406	190; 100;
	ST	Perdite	Time
0	1	0,0672698	100; 70;
5	0	0,06949305	120; 100;

		tempo exe	00:00:13.7127843
	ST	Perdite	Time
0	2	0,0084096	100; 70;
5	2	0,0127405	70; 100;
	ST	Perdite	Time
0	0	0,0427921	100; 30;
5	3	0,0545117	40; 100;
	ST	Perdite	Time
0	1	0,1312805	100; 120;
5	2	0,1270509	270; 100;
	ST	Perdite	Time
0	1	0,0672698	100; 70;
5	0	0,0694931	120; 100;
	ST	Perdite	Time
0	6	0,1251273	100; 220;
5	3	0,1214281	90; 100;
	ST	Perdite	Time
0	1	0,0963884	100; 130;
5	2	0,0939318	190; 100;
	ST	Perdite	Time
0	1	0,0946127	100; 110;
5	2	0,0943987	210; 100;
	ST	Perdite	Time
0	1	0,03114	100; 70;
5	5	0,0129374	190; 100;
	ST	Perdite	Time
0	1	0,0672698	100; 70;
5	0	0,0694931	120; 100;

Figura 4.9: Risultati: 6 Nodi 2 Generatori

Bibliografia

[1] Marco Dorigo, Thomas Stützle, *Ant Colony Optimization*, 2004

[2] James McCaffrey, *Test Run - Ant Colony Optimization*, 2012

[3] J. A. Michline Rupa, S. Ganesh, *Power Flow Analysis for Radial Distribution System Using Backward/Forward Sweep Method* , 2014

[4] http://www.carlovecchio.altervista.org/c—-permutazioni-e-combinazioni.html

[5] http://www.koderdojo.com/blog/depth-first-search-algorithm-in-csharp-and-net-core

[6] MathNet

[7] https://www.codeproject.com/Articles/43438/Connect-C-to-MySQL