UNIVERSITÀ POLITECNICA DELLE MARCHE

Corso di laurea magistrale in
INGEGNERIA ELETTRONICA

Implementazione ed ottimizzazione di comunicazioni multicast in reti ToLHnet

Implementation and optimization of multicast communications in ToLHnet protocol

Tesi di Laurea di

Riccardo LAIOLO

Relatore

Giorgio BIAGETTI

Correlatore

Simone ORCIONI

ANNO ACCADEMICO 2018/2019

Indice

Elenco delle figure . . . III

1 Introduzione **4**

1.1 Stato dell'arte . . . 5

1.2 Ambiente di lavoro . . . 10

2 Protocollo ToLHnet **11**

2.1 Modalità di indirizzamento e routing . . . 13

2.2 Nuove modalità di indirizzamento proposte . . . 15

2.3 Implementazione del firmware di riferimento . . . 16

2.3.1 Regole di filtraggio . . . 16

2.3.2 Gruppi . . . 18

2.3.3 Configurazione . . . 19

3 Generazione tabelle di routing multicast **23**

3.1 Principio di creazione dei percorsi . . . 23

3.2 Algoritmi genetici . . . 25

3.2.1 Inizializzazione . . . 25

3.2.2 Selezione . . . 26

3.2.3 Incrocio . . . 26

3.2.4 Mutazioni . . . 26

3.2.5 Interruzione della ricerca . . . 27

3.3 Algoritmo implementato . . . 27

3.3.1 Funzione di fitness . . . 28

3.3.2 Generazione della popolazione iniziale . . . 34

3.3.3 Selezione . . . 36

3.3.4 Crossover . . . 38

3.3.5 Mutazioni . . . 43

3.3.6 Struttura dell'algoritmo . . . 47

4 Analisi dei risultati **52**
4.1 Analisi della generazione dei percorsi 52
4.2 Simulazione del packet loss 56

5 Conclusioni **61**

Appendice **62**

A Generatore pseudorandom con PDF non uniforme **62**

B Struttura dei dati utilizzati nell’algoritmo genetico **63**

Riferimenti bibliografici **68**

Elenco delle figure

2.1 Esempio di una possibile topologia di rete 11
2.2 Esempio di un possibile albero logico 12
2.3 Struttura di un pacchetto a livello rete 13
2.4 Algoritmo di filtraggio 14
2.5 Struttura dei possibili payload di configurazione 19
2.6 Payload di esempio per la configurazione dei gruppi 21
3.1 Esempio di un percorso di broadcast 24
3.2 Crossover 27
3.3 Connessioni uscenti 28
3.4 Reciproco del PER 30
3.5 funzione $g(r)$ eq.(3.4) 31
3.6 Popolazione iniziale 36
3.7 Operazione modulo 37
3.8 Crossover 1 39
3.9 Crossover 2 41
3.10 Mutazione 1 44
3.11 Mutazione 2 45
4.1 Reti quadrate 52
4.2 Rete circolare con 1000 nodi 53
4.3 Reti poligonali 54
4.4 Domini di multicast di prova 55
4.5 Percorso analizzato 56
4.6 PCR cumulativa 57
4.7 PCR cumulativa con correzione errori 58
4.8 PCR cumulativa percorso con salti più brevi 59
A.1 PMF e CDF discrete 62

1 Introduzione

Il lavoro presentato in questa tesi consiste nell'estensione del protocollo di comunicazione ToLHnet [1] al supporto di pacchetti di tipo multicast e broadcast non precedentemente supportati, quindi nello sviluppo del firmware per la piattaforma d'esempio ed infine nello sviluppo delle procedure C++ per il calcolo dei percorsi di broadcast e multicast da implementare nel codice del nodo master.

Il protocollo ToLHnet è utile quando si necessita di implementare una rete di sensori e attuatori. Questi possono comunicare in maniera trasparente attraverso mezzi trasmissivi differenti tramite una rete logica strutturata ad albero. Nella sezione 2 verrà descritto il protocollo in maniera più approfondita.

I percorsi sono una serie di nodi che il pacchetto deve attraversare e, dai quali, viene ritrasmesso. Una volta che il pacchetto ha raggiunto l'ultimo nodo l'informazione è stata propagata a tutti i nodi della rete (o ad un sottoinsieme nel caso di multicast). Non è necessario che il percorso attraversi ogni singolo nodo da informare poiché, quando il pacchetto viene ritrasmesso, questo viene ricevuto, oltre che dal nodo successivo nel percorso, anche da tutti i nodi connessi allo stesso bus che sono sufficientemente vicini. L'algoritmo implementato per il calcolo dei percorsi, come si vedrà nella sezione 3, è di tipo genetico, ovvero una procedura iterativa ispirata ai meccanismi biologici di evoluzione della specie.

In un protocollo di comunicazione il supporto a trasmissioni broadcast e multicast è utile quando si deve inviare uno stesso comando ad una moltitudine di nodi. Per esempio se si vuole accendere o spegnere l'illuminazione in un'intera ala di un edificio, o magari per gestire dinamicamente l'illuminazione stradale in funzione della presenza di utenti della strada.

Si vedrà, per prima cosa, un breve excursus sulle tipologie di protocolli di routing per le reti di sensori presenti in letteratura. A seguire ci sarà prima una descrizione del protocollo ToLHnet e le nuove modalità di instradamento proposte, quindi verranno descritte le modifiche apportate al firmware al fine di implementare le nuove modalità di instradamento. Ci sarà poi una parte dedicata agli algoritmi genetici in generale e, a seguire, la descrizione dell'algoritmo sviluppato. Durante la trattazione si parlerà anche, quando rilevante, delle difficoltà e dei problemi incontrati e delle soluzioni adottate durante le varie fasi dello sviluppo. Verranno infine analizzati i risultati ottenuti al fine di valutare le prestazioni del

protocollo in uno scenario implementativo realistico.

1.1 Stato dell'arte

I protocolli di instradamento nelle reti di sensori senza fili possono essere suddivisi in tre categorie sulla base dell'architettura di rete su cui devono operare: instradamento flat, gerarchico o basato sulla posizione dei nodi. Questi protocolli possono essere inoltre classificati in base alla modalità principale con cui opera il protocollo [2].

Tipicamente, una rete di sensori è composta da centinaia o migliaia di nodi. Questi possono comunicare tra loro o direttamente con una stazione base esterna. Questi nodi sono disseminati a coprire l'area di misura e ognuno di questi nodi ha la capacità di raccogliere dati o di instradare i dati altri nodi agli altri sensori o alla stazione base.

L'instradamento dei pacchetti nelle reti di sensori è impegnativo per le caratteristiche intrinseche di queste tipologie di reti: a causa del numero relativamente alto di nodi non è possibile impiegare uno schema di indirizzamento globale poiché questo aumenterebbe in maniera inaccettabile l'overhead; tipicamente, inoltre, il flusso di informazioni in una rete di sensori origina, principalmente, da diversi nodi e si estingue presso la stazione base; in più i nodi possono essere fortemente limitati in termini di capacità di calcolo, di memoria e di energia. Inoltre i dati raccolti dai singoli nodi possono possono essere fortemente correlati gli uni dagli altri, gli algoritmi di instradamento devono tenere conto quindi della ridondanza dei dati per per ridurre il costo delle trasmissioni in termini di banda e di energia. Infine, tipicamente in una rete di sensori non vengono interrogati direttamente i singoli nodi, ma piuttosto si interroga l'intera rete su una certa tipologia di informazione: per esempio, se la stazione base chiede alla rete informazioni riguardanti la temperatura in un certo range, tutti i nodi che dispongono di questa informazione faranno convergere il dato richiesto verso la stazione secondo le procedure caratteristiche del particolare algoritmo.

A causa di queste differenze non è possibile utilizzare efficientemente i protocolli comuni per le reti wireless. Per minimizzare il consumo energetico gli algoritmi di instradamento per le reti di sensori fanno uso di diverse strategie, alcune già note, altre appositamente studiate, come il data aggregation, l'in-network processing, il clustering, l'uso di nodi con ruoli diversificati e l'uso di metodi

data-centrici. Inoltre questi protocolli possono essere distinti in: basati sul multipath, basati sulle richieste e basati sulla qualità del servizio a seconda delle modalità con cui il protocollo gestisce la rete. Nelle reti flat i nodi sono tutti uguali e svolgono gli stessi ruoli, mentre nei protocolli gerarchici si cerca di raggruppare i nodi in gruppi in modo da poter applicare una prima aggregazione e riduzione ai dati. Nei protocolli basati sulla localizzazione dei nodi viene sfruttata la conoscenza della posizione per inoltrare le informazioni solo dove necessario invece che a tutta la rete.

Uno degli obiettivi principali dei protocolli di instradamento per le reti di sensori è quello di portare a destinazione l'informazione cercando di massimizzare il periodo di vita della rete, evitando allo stesso tempo di impattare sulla qualità delle connessioni, applicando delle tecniche di gestione energetica. Lo sviluppo e la progettazione di un protocollo di instradamento per reti di sensori deve tenere in considerazione tutta una serie di fattori che rendono estremamente vari i possibili scenari. Il dislocamento dei nodi dipende dall'applicazione specifica, e può essere nota a priori o random, quindi i percorsi lungo i quali i dati devono essere indirizzati possono essere precalcolati oppure determinati dinamicamente. I nodi devono ottimizzare il proprio consumo energetico poiché, in una rete multihop, i nodi, oltre a svolgere il compito di sensori, svolgono il ruolo di router per i dati provenienti dagli altri nodi, quindi se un nodo si spegne ci possono essere conseguenze significative sulla topologia della rete. A seconda del tipo di applicazione della rete i singoli nodi possono riportare i dati su base temporale, in seguito a eventi particolare, in seguito a richieste o in una combinazione di queste modalità. I vari protocolli di instradamento devono quindi essere a prova di guasto e mantenere la rete operativa anche se alcuni nodi smettono di rispondere; devono funzionare in reti diverse con una grande variabilità sul numero di nodi; dovrebbero poter tollerare che i nodi godano di una certa mobilita e non siano immobili; dovrebbero implementare inoltre sia delle strategie di aggregazione dei dati per ridurre il numero di trasmissioni e che dei parametri di QoS per garantire il recapito dei dati entro il tempo utile.

Reti flat

Vediamo ora una rapida descrizione di alcuni protocolli di instradamento presenti in letteratura. Nelle reti flat, dato il grande numero di nodi, non è possibile assegnare un identificativo globale ad ogni singolo nodo, questo ha portato allo

sviluppo di protocolli con instradamento data-centrico, dove la stazione base richiede alla rete, o ad una regione di essa, una certa tipologia di dati e attende che questi gli vengano recapitati. Due protocolli di questo tipo sono: *Sensor Protocols for Information via Negotiation - SPIN* e *Directed diffusion*.

SPIN è una famiglia di protocolli nei quali le trasmissioni sono governate da negoziazioni e ogni nodo invia i propri dati a tutti i nodi presupponendo che ogni altro nodo potrebbe essere la stazione base. Questo permette di ricevere l'informazione richiesta interrogando qualsiasi nodo della rete. Per ridurre la quantità di informazioni che vengono trasmesse i nodi non trasmettono per intero ogni dato a disposizione, ma informa i nodi vicini trasmettendo i metadati relativi ai dati pronti per la trasmissione, i nodi vicini analizzano questi metadati ed eventualmente richiedono le informazioni di cui necessitano. I protocolli della famiglia SPIN riducono la quantità di dati che viaggiano nella rete rispetto ai protocolli tradizionali che fanno uso di strategie di flooding, riducendo quindi l'energia e la banda sprecata trasmettendo dati inutili o copie non richieste. Un vantaggio dei protocolli SPIN è che le variazioni nella struttura della rete a seguito di variazioni topologiche sono circoscritte in quanto ogni nodo deve conoscere solo i nodi che distano da lui un singolo salto. D'altro canto però, il meccanismo di annuncio e negoziazione dei metadati non garantisce la diffusione dei dati.

L'idea alla base del paradigma *Directed diffusion* è di combinare i dati provenienti da diverse fonti durante il percorso verso la stazione base, riducendo la ridondanza e minimizzando il numero di trasmissioni. La stazione base richiede informazioni annunciando un "interesse". Questo viene propagato lungo la rete da ogni nodo verso i vicini e ognuno aggiorna il gradiente di interesse nella direzione dalla quale lo riceve. L'obiettivo è quello di trovare un buon albero di aggregazione che porti i dati di interesse dalla fonte verso il primo punto da cui è originato l'interesse. In SPIN, i dati vengono "offerti" dalle sorgenti quando disponibili, in Directed diffusion, invece, i dati vengono richiesti quando necessari.

Reti gerarchiche

I protocolli di instradamento di tipo gerarchico sono stati introdotti e sono largamente utilizzati nelle reti cablate. La ragione è che questi hanno grandi vantaggi nella scalabilità della rete e nell'efficienza delle trasmissioni. Nelle reti gerarchiche i nodi sono raggruppati su base geografica in piccoli sottoinsiemi; in ogni

gruppo è presente un nodo più importante degli altri, cioè dotato di maggiori potenzialità, il cui compito è quello di aggregare i dati, processarli e trasmetterli alla stazione base. I nodi sensore, che dispongono di risorse limitate, invece, devono acquisire i dati e farli convergere ai nodi router di zona. L'instradamento gerarchico è principalmente composto da due strati: uno per la selezione del nodo principale di ogni gruppo, l'altro per la gestione dell'instradamento.

Uno dei primi algoritmi appartenenti a questa famiglia è il *Low Energy Adaptive Clustering Hierarchy (LEACH)*. In questo protocollo è prevista la generazione dinamica dei cluster: alcuni nodi tra quelli dell'intera rete (circa il 5 %) vengono scelti casualmente per essere i componenti principali dei cluster e, periodicamente, viene rifatta l'estrazione per distribuire il consumo energetico. Il nodo principale aggrega e comprime i dati provenienti dai nodi del cluster e li invia alla stazione base. Poiché in questo protocollo la raccolta dei dati è centralizzata e viene fatta su base temporale, questa strategia è idonea in tutte quelle applicazioni dove è necessario un campionamento continuo dell'ambiente. Benché sia dimostrato che LEACH estenda il periodo di vita della rete, l'estrazione random dei nodi principali presuppone che ogni nodo sia in grado di comunicare direttamente con la stazione base e che ognuno sia sufficientemente potente da supportare più schemi di accesso al mezzo (protocolli MAC). Pertanto, il protocollo LEACH non è applicabile in reti che si estendono su superfici significative e presuppone che nodi vicini presentino dati correlati tra loro in modo che sia possibile applicare in maniera efficace le strategie di aggregazione dei dati. Infine, per distribuzioni di nodi non uniformi, poiché l'estrazione dei nodi principali è casuale, è possibile che, nelle zone con meno nodi principali, i cluster abbiano dimensioni talmente grandi da vanificare l'effetto di aggregazione e compressione, in termini di consumo energetico della rete. In letteratura sono descritti molti protocolli che vanno a migliorare e risolvere queste problematiche, ma poiché la descrizione di questi è oltre lo scopo di questa trattazione si rimanda a [2].

Routing location-based

In questa famiglia l'indirizzamento viene fatto tramite le posizioni dei nodi, ogni nodo deve quindi conoscere la propria. Le distanze tra i nodi possono essere stimate valutando la potenza ricevuta e le coordinate relative dei singoli nodi possono essere ottenute scambiando tra i nodi le distanze stimate. In alternativa i nodi possono essere equipaggiati con ricevitori GPS per avere la posizione esat-

ta. Le modalità di instradamento sono simili a quelle dei protocolli gerarchici, ma la conoscenza della posizione dei nodi e la possibilità di indirizzarli utilizzando le coordinate permette di propagare le richieste solo nelle regioni dove queste verranno poi effettivamente processate, minimizzando il numero di trasmissioni inutili. Inoltre, permette di spegnere selettivamente delle regioni di rete, o un sottoinsieme di nodi in una stessa regione, per minimizzare il consumo della rete mantenendo comunque la possibilità di instradare i pacchetti.

Benché la maggior parte dei protocolli di instradamento rientrino in una delle categorie descritte, alcuni affrontano il problema dal punto di vista del flusso di rete e altri dal punto di vista del QoS [3]. Il primo approccio viene impiegato in *Maximum Lifetime Energy Routing*, dove il periodo di funzionamento della rete viene massimizzato definendo il costo di ogni collegamento in funzione dell'energia residua del nodo e di quella richiesta dalla trasmissione usando quel particolare collegamento. Una volta calcolati tutti i pesi viene calcolato il percorso a costo minore secondo l'algoritmo di Bellman-Ford, quindi il percorso trovato è quello che massimizza l'energia residua della rete dopo aver concluso la trasmissione. *Sequential Assignment Routing* è il primo protocollo in cui sono stati inclusi aspetti di QoS. Il protocollo SAR crea degli alberi di instradamento che originano dai nodi distanti un salto dal destinatario delle informazioni tenendo in considerazione l'energia di ogni percorso, la priorità del pacchetto e il QoS. Questi alberi vengono utilizzati per formare dei percorsi multipli che uniscono la sorgente e la destinazione dell'informazione. L'informazione viene infine trasmessa sul percorso migliore in termini di energia e QoS. I percorsi multipli vengono mantenuti in memoria per rendere il protocollo robusto nei confronti di malfunzionamenti localizzati. Per reti particolarmente grandi questo protocollo soffre di un grosso overhead dovuto al mantenimento di tutte le tabelle di instradamento e gli stati dei nodi.

Come si è visto, non esiste una soluzione unica in quanto il problema da risolvere presenta, in ogni applicazione, caratteristiche diverse: se si vuole monitorare una regione al fine di individuare eventuali intrusioni, potrebbe essere opportuna una strategia dove sono i nodi sensori ad informare la rete degli eventi che hanno rilevato; se invece si vuole implementare una rete di sensori per il campionamento ambientale (qualità dell'aria, temperatura, ...), potrebbe essere

migliore un approccio inverso dove è la stazione base a richiedere alla rete, su base temporale, di farsi recapitare i campioni acquisiti. Se invece la rete non è schematizzabile con un solo sensore "diffuso", ma è fondamentale mantenere l'individualità di ogni nodo, per esempio nel caso di una rete in cui sono presenti anche attuatori o dove i dati che originano dai vari nodi godono di una scarsa ridondanza e sovrapposizione, sarebbe più opportuno un protocollo di impronta più classica dove ogni nodo ha un proprio identificativo. Il protocollo ToLHnet rientra in quest'ultima categoria: ogni nodo ha un proprio indirizzo e lo schema di indirizzamento principale è di tipo unicast.

1.2 Ambiente di lavoro

Lo sviluppo è stato eseguito in ambiente Linux (distribuzione Linux Mint 19.2) utilizzando, come editor di codice sorgente, *`Visual Studio Code`*. Per la compilazione, sia per il firmware che per il demone master, è stato usato il compilatore del progetto GNU *`GCC`*. In particolare: per la crosscompilazione del firmware dei nodi periferici, la versione *`arm-none-eabi-gcc 7.3.1`* presente nel repository PPA *`team-gcc-arm-embedded`* [4]; mentre per la compilazione del codice del nodo master la versione presente nei repository di sistema (*`gcc 7.4.0`* al momento della scrittura).

Inoltre, per la compilazione del firmware, è necessario il pacchetto *TivaWare™ for C Series* scaricabile dal sito della Texas Instruments [5], mentre per la compilazione del codice master sono necessarie le librerie *Boost C++* presenti nei repository di sistema (*`libboost-all-dev`* versione *`1.65.1.0`* al momento della scrittura)

2 Protocollo ToLHnet

Il protocollo ToLHnet (Tree or Linear Hopping network) permette la creazione di reti miste composte da nodi con capacità di calcolo più o meno limitate. Delle possibili implementazioni di questo protocollo potrebbero essere il monitoraggio e il controllo di sistemi di illuminazione pubblica, di impianti domotici o comunque di tutte quelle applicazioni dove si rende necessaria l'intercomunicazione di sensori e attuatori.

Il protocollo è stato sviluppato con l'idea di mantenere più bassi possibile la complessità del firmware dei nodi e l'overhead introdotto dal livello di rete. Quindi la complessità dell'implementazione è sbilanciata verso un singolo nodo dotato di una potenza di calcolo maggiore, che svolge il ruolo di controllore della rete.

Le caratteristiche più importanti sono:

- supporto trasparente a mezzi trasmissivi di varia natura, cablati e non;
- indirizzi a 16bit (supporto per oltre 60000 nodi);
- overhead di rete minore del 2 % (4 byte di intestazione e payload fino a 240 byte)
- bassa complessità del firmware dei nodi (< 12 kB)

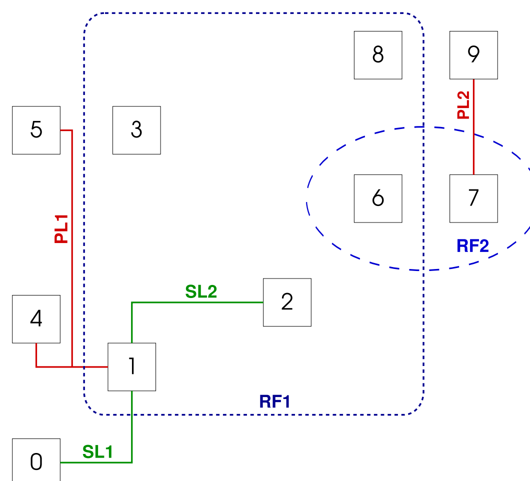


Figura 2.1: *Esempio di una possibile topologia di rete*

Da una descrizione di tutte le interconnessioni fisiche presenti tra i nodi che formano la rete (fig. 2.1) viene estratto un albero logico minimo che origina dal nodo master (fig. 2.2). I pacchetti possono spostarsi nella rete solo lungo i rami dell'albero logico.

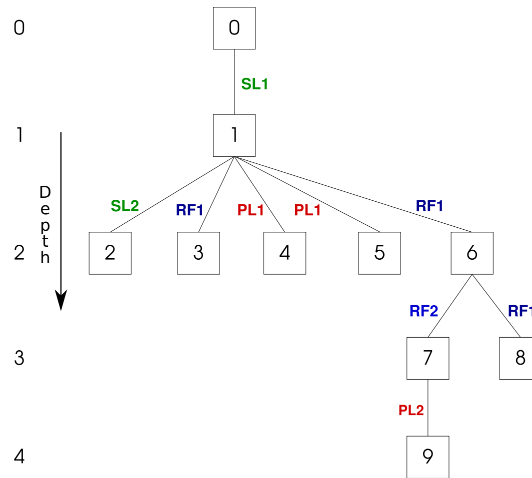


Figura 2.2: *Esempio di un possibile albero logico*

Idealmente, in una struttura ad albero, le strategie necessarie per il corretto instradamento dei pacchetti sono molto più semplici rispetto ad una topologia magliata. A causa della natura eterogenea dei possibili collegamenti fisici, in particolare la possibile presenza di connessioni non punto-punto, i nodi possono ricevere dei pacchetti anche da connessioni che, benché presenti nella descrizione fisica delle interconnessioni, non fanno parte dell'albero logico o addirittura da connessioni non previste dalla descrizione fisica della rete. Nell'albero di fig. 2.2, per esempio, i nodi *1*, *3*, *6* e *8* sono connessi ad uno stesso bus wireless, però, poiché il nodo *8* è più distante degli altri dal nodo *1* deve sfruttare il nodo *6* come ripetitore. Si supponga invece che i nodi *3* e *6* siano sufficientemente vicini da poter ascoltare l'uno i pacchetti inviati dall'altro. Il nodo *6* riceverebbe involontariamente anche i pacchetti destinati a *3*, mentre *3* riceverebbe sia quelli destinati a *6* che quelli che vengono inoltrati da *6* verso *8*. L'eventuale processamento o l'inoltro di questi pacchetti potrebbe addirittura portare all'avaria dell'intera infrastruttura di rete: in reti più complesse di quella di fig. 2.2 potrebbero crearsi degli anelli in cui un pacchetto viene inoltrato indefinitamente. Quindi i nodi devono essere in grado di riconoscere, sulla base delle informazioni contenute

nell'intestazione del pacchetto, se questo deve essere effettivamente processato o se deve essere scartato.

2.1 Modalità di indirizzamento e routing

Nell'ultima versione del protocollo ToLHnet sono utilizzate due modalità di indirizzamento, cioè utilizzando l'indirizzo hardware del nodo di destinazione o utilizzando il suo indirizzo di rete. L'indirizzo hardware (MAC address) è lungo 6 byte e viene solitamente assegnato in fase di produzione e identifica il particolare hardware, l'indirizzo di rete invece viene assegnato al nodo in fase di inizializzazione della rete ed è di 2 byte. L'indirizzamento tramite MAC address viene usato solo durante l'inizializzazione della rete, cioè quando non sono ancora stati assegnati gli indirizzi di rete ai vari nodi.

In fig. 2.3 è riportata la struttura di un pacchetto a livello rete. Il campo `AM` serve a determinare il tipo di indirizzamento con cui il pacchetto è stato formattato, ovvero determina quali dei campi facoltativi sono presenti.

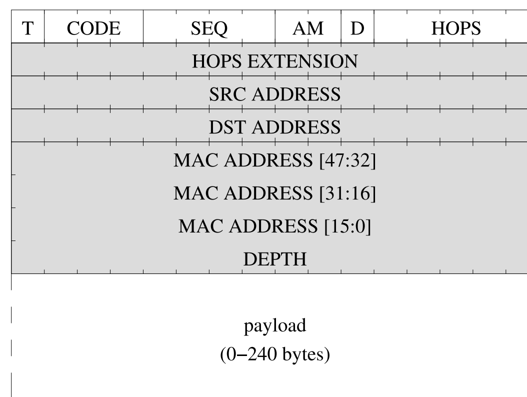


Figura 2.3: *Struttura di un pacchetto a livello rete*

Un nodo, per determinare se accettare, inoltrare o scartare un pacchetto che ha ricevuto deve applicare tutta una serie di regole e criteri. queste regole sono memorizzate in una struttura che si chiama tabella di instradamento (dall'inglese routing table, forwarding table o forwarding information base)

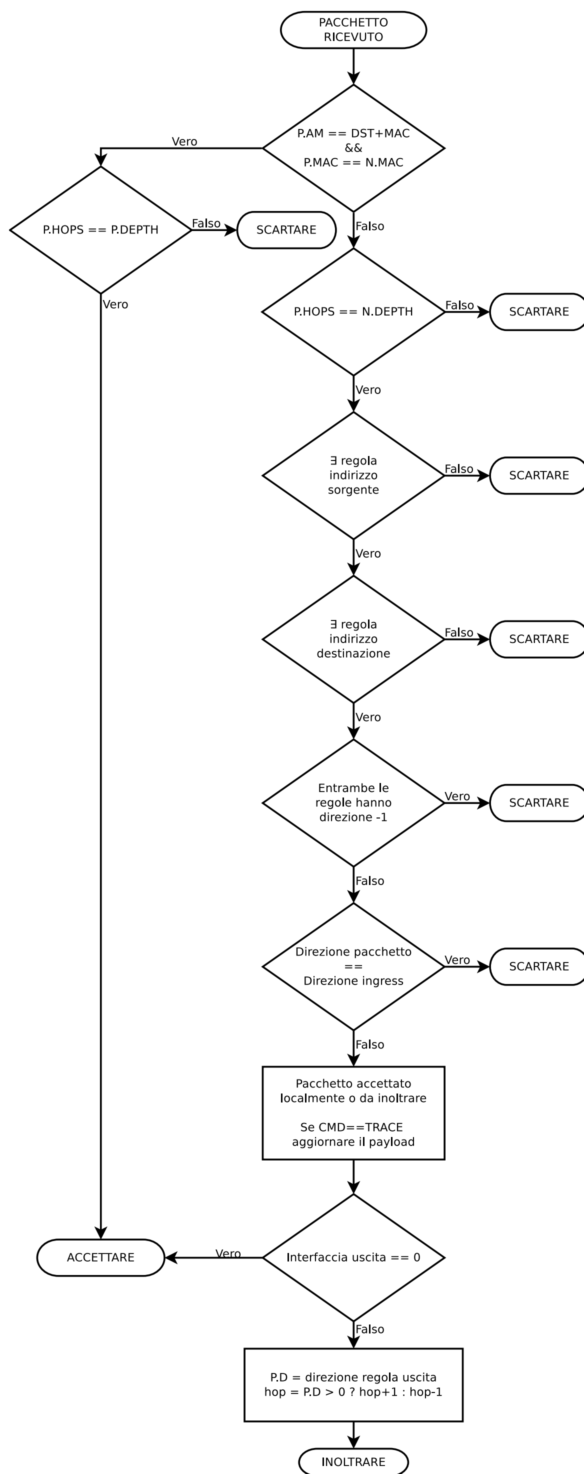


Figura 2.4: *Algoritmo di filtraggio*

Nello schema in fig. 2.4 è riportato l'algoritmo che viene applicato ad ogni pacchetto ricevuto. Per una descrizione dettagliata si rimanda alla documentazione ufficiale del protocollo [6]. Con i prefissi P e N nello pseudo codice contenuto nei blocchi del diagramma si intendono rispettivamente i dati che sono contenuti nell'intestazione del pacchetto ricevuto e i dati già presenti nella memoria del nodo. Questo set di istruzioni gestisce senza errori i pacchetti che si muovono lungo i rami dell'albero, scartando tutti quei pacchetti che arrivano da connessioni non tracciate nell'albero o che percorrono rami tracciati nella direzione sbagliata. Questo algoritmo però non è adatto alla gestione dei pacchetti multicast poiché questi non percorreranno generalmente connessioni presenti nell'albero e quindi verrebbero scartati come pacchetti ascoltati involontariamente.

Si noti che, allo stato attuale delle specifiche del protocollo, il valore di profondità del nodo nell'albero della rete

dipende solo dal nodo.

Le regole che vengono cercate al terzo e al quarto passo dell'algoritmo sono presenti nella memoria del nodo e gli vengono fornite dal nodo master in fase di inizializzazione della rete insieme al proprio indirizzo e la propria profondità nell'albero. Queste regole sono sostanzialmente un elenco di indirizzi di rete (o di intervalli di indirizzi) associati ognuno alla periferica verso la quale reindirizzare il pacchetto. Per fare un esempio si riprenda l'albero in fig. 2.2: il nodo `1` avrà in memoria sicuramente le seguenti voci nella sua tabella di routing:

- Dest: 2 ⇒ SL2
- Dest: 3 ⇒ RF1
- Dest: [4-5] ⇒ PL1
- Dest: [6-9] ⇒ RF1

oltre a queste regole, ogni nodo aggiungerà due regole di default: quelle per l'accettazione locale del pacchetto, ovvero quando l'indirizzo di destinazione coincide con il proprio indirizzo e la regola che instrada i pacchetti in direzione del suo nodo genitore. A sua volta il nodo `6` avrà delle regole simili per smistare i pacchetti tra i due rami che originano da esso.

2.2 Nuove modalità di indirizzamento proposte

Si è pensato di riservare una porzione dello spazio degli indirizzi alle comunicazioni di tipo broadcast e multicast: gli indirizzi `0xF000-FFFF`, ovvero quelli che hanno il nibble più significativo pari a `0xF`, indicano destinazioni multicast, la seconda cifra indica quale dominio multicast è il destinatario, mentre le ultime due cifre indicano quale gruppo deve accettare il pacchetto. Quindi è possibile definire 16 domini di multicast e 256 gruppi per un totale di 4096 indirizzi riservati. Il dominio `0xF` è riservato e non è definibile dall'utente e ne fanno parte tutti i nodi. Analogamente il gruppo `0xFF` non è assegnabile dall'utente e ogni nodo ne accetta i relativi pacchetti. Quindi l'indirizzo `0xFFFF` assume il significato speciale di indirizzo di broadcast. Ovviamente il supporto a comunicazioni di tipo multicast implica un restringimento dello spazio degli indirizzi dall'intervallo `0x0000-FFFF` a `0x0000-EFFF` (61440 possibili indirizzi).

2.3 Implementazione del firmware di riferimento

Si discuteranno le modifiche apportate al firmware di riferimento (microcontrollore Texas Instruments TM4C123G con core ARM Cortex-M4) per estendere l'implementazione al supporto di pacchetti multicast e ai gruppi. Per mantenere il protocollo semplice e il firmware poco costoso in termini di risorse hardware si è cercato di modificare gli algoritmi esistenti il minimo possibile.

A ogni dominio di multicast è associato un percorso che il pacchetto deve percorrere per raggiungere ogni nodo del dominio. Poiché il percorso viene calcolato sulla base della descrizione fisica della rete minimizzando una qualche funzione costo, questo potrebbe unire i nodi in maniera completamente diversa dall'albero logico. Bisogna quindi implementare delle nuove regole di filtraggio, che permettano a questi pacchetti speciali di spostarsi lungo la rete, e una logica per la gestione dei gruppi. Questo comporta quindi anche una modifica alla procedura iniziale di configurazione del nodo.

2.3.1 Regole di filtraggio

Si è pensato di utilizzare la tabella di forwarding già presente per l'indirizzamento dei pacchetti unicast per l'indirizzamento dei pacchetti di multicast. Poiché un nodo potrebbe essere attraversato più volte (in direzioni diverse) da uno stesso percorso di multicast (per permettere al percorso di raggiungere tutti i nodi interessati senza ricorrere a diramazioni e quindi per assicurare che nella rete transiti al più un solo pacchetto) non basta il solo indirizzo di destinazione per determinare il comportamento del nodo, ma occorre anche valutare il numero di salti che il pacchetto ha compiuto nella rete. Si è quindi modificata l'implementazione delle tabelle di forwarding:

```
struct fib_entry
{
  uint16_t    net;
  uint16_t    mask;
  uint8_t     mask_length;
  uint8_t     type;
  int8_t      direction;
  char        device;
  uint8_t     accept_flag;
  uint16_t    depth;
};
```

Sono stati aggiunti i campi *accept_flag* e *depth*. Il primo serve per determinare, unitamente a *device*, se il pacchetto deve essere inoltrato, accettato o entrambi, cioè il nodo potrebbe venir attraversato dal pacchetto anche se non fa parte del dominio di multicast (solo inoltro), potrebbe far parte del dominio ma non venire attraversato dal percorso (solo ricezione) o potrebbe far parte sia del dominio che del percorso (sia ricezione che inoltro). Il secondo serve per poter scartare il pacchetto qualora venisse ascoltato da comunicazioni vicine e per distinguere le regole per diversi passaggi del pacchetto. Si sarebbe potuto pensare, poiché *device* è un valore a 7 bit e *accept_flag* è 1 bit, di utilizzare i bit field nella definizione della struttura, ma dato che l'architettura di riferimento è a 32 bit questo non avrebbe portato alcun miglioramento poiché si sarebbe ridotta la dimensione della struttura da 11 byte a 10 byte che, in entrambi i casi, occupano 3 word di memoria.

Per estrarre dalla tabella di forwarding una regola occorre valutare l'indirizzo e la profondità del nodo nell'albero di instradamento, è stata definita allora la nuova funzione di ricerca *fib_lookup_2*. Poiché la funzione di ricerca viene utilizzata in altri punti del codice, la funzione originale *fib_lookup* non è stata sostituita, però è stata modificata in modo da chiamare la nuova funzione passandogli, come valore di profondità, quella del nodo nell'albero di unicast così da non avere due funzioni distinte che sono l'una un caso particolare dell'altra e risparmiare quindi un po' di spazio in memoria.

```
struct fib_entry const *fib_lookup (uint16_t address)
// Old implementation for compatibility
{
  return fib_lookup_2 (address, fib_table[0].depth);
}

struct fib_entry const *fib_lookup_2 (uint16_t address, uint16_t hops)
{
  unsigned i;
  for (i = 0; i < fib_size; ++i)
  if (
  (fib_table[i].type == 0 && fib_table[i].net == (address &
   ↪  fib_table[i].mask) && fib_table[i].depth == hops)
  ||
  (fib_table[i].type == 1 && address >= fib_table[i].net && address <=
   ↪  fib_table[i].mask && fib_table[i].depth == hops)
  ) {
    return fib_table + i;
  }
  return NULL;
}
```

A questo punto il nodo dispone di tutte le informazioni necessarie per permettere la corretta gestione dei pacchetti, si è modificato quindi l'algoritmo di filtraggio descritto in [6] e riportato in fig. 2.4, implementato nella funzione `packet_received` nel file `network.c`, come segue: è stato eliminato il controllo sulla profondità del nodo nell'albero poiché questa non è più una proprietà globale del nodo, ma delle singole regole di forwarding; sono state scambiate, nell'ordine dei controlli eseguiti, la ricerca della regola per la destinazione e della regola per la sorgente, questo perché la causa più probabile che porta un pacchetto a venire scartato è la mancata corrispondenza del campo `hops` con la profondità del nodo nella rete, quindi conviene controllare per prima l'esistenza della regola sulla destinazione. Il resto dell'algoritmo è sostanzialmente invariato rispetto a quello riportato nelle specifiche del protocollo tranne che nella parte finale, poiché l'inoltro e l'accettazione locale del pacchetto non più sono mutuamente esclusivi.

2.3.2 Gruppi

L'implementazione dei gruppi è utile per suddividere ulteriormente l'insieme dei nodi della rete senza però aumentare la quantità di possibili percorsi di multicast (i domini di multicast potrebbero essere utilizzati per separare la rete in zone geograficamente distinte, mentre i gruppi per distinguere i nodi in base alla loro funzione). Un pacchetto destinato al gruppo *m-esimo* dell' *n-esimo* dominio di multicast verrà recapitato a tutti i nodi del dominio *n*, ma verrà accettato e processato solo dai nodi facenti parte del gruppo *m*. Implementare questo comportamento è relativamente semplice: basta controllare, prima di passare il pacchetto all'application layer, se il nodo fa parte del gruppo *m*.

```
if (h.target) {
  if (h.multicast)
  {
    uint8_t mc_domain = (uint8_t)((h.dst & 0X0F00)>>8), mc_group =
     ↪  (uint8_t)(h.dst & 0X00FF);
    if (!grp_lookup(mc_domain,mc_group)) return;
  }
  // send the packet to the transport layer:
  if (datagram_received) datagram_received(h.src, h.seq, h.code, data,
   ↪  len, h.multicast);
}
```

Come si può vedere, se il pacchetto è destinato all'application layer viene controllata anche l'appartenenza del nodo al gruppo tramite la funzione *grp_lookup*.

I gruppi sono memorizzati in maniera simile alle regole di forwarding. Un array di strutture *grp_entry* contiene le coppie dominio/gruppo, inoltre sono state scritte le funzioni per inserire un elemento (*grp_add*), eliminare un elemento (*grp_del*) e per eliminare tutti gli elementi relativi ad uno stesso dominio (*grp_del_dom*)

2.3.3 Configurazione

Ora che sono state implementate tutte le strategie di instradamento occorre implementare la configurazione dei nodi, ovvero ideare la formattazione dei pacchetti di configurazione per le nuove informazioni necessarie e il relativo parsing.

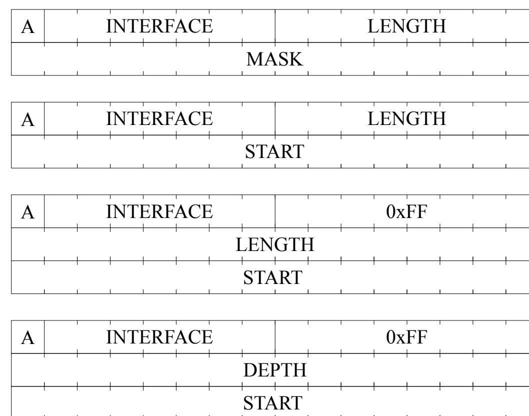


Figura 2.5: *Struttura dei possibili payload di configurazione*

In fig. 2.5 sono riportati le strutture dei pachetti di configurazione, le prime tre sono le strutture per le regole di unicast già previste in [6] con l'unica differenza che il primo bit del primo byte indica se i pacchetti che ricadono in quella regola devono essere solo inoltrati o anche accettati. Per le regole di unicast non ci sono situazioni previste in cui questo bit non sia 0, e poiché l'interfaccia è identificata da un carattere ASCII (valore a 7-bit allineato a destra), la presenza o meno di questo bit non altera, in pratica, l'implementazione precedente. L'ultima invece è la nuova struttura per le regole multicast, queste sostanzialmente sono delle

regole di tipo span dove però l'estensione dell'intervallo è fissa (*0xFF*), quindi non occorre trasmetterla, al suo posto viene trasmesso il valore di profondità.

Sono state modificate le funzioni *fib_add* e *fib_add_span* in modo da scrivere correttamente le nuove informazioni aggiunte nella struttura *fib_entry*. Di seguito sono riportate le righe aggiunte alla prima funzione, per la seconda sono state apportate le stesse modifiche:

```
bool fib_add (uint16_t net, uint8_t length, char device, int direction,
 ↪  uint16_t depth)
{
[...]
  fib_table[i].device = device & 0x7F;
  fib_table[i].accept_flag = device==0 ? 1 : device >> 7;
  fib_table[i].depth = depth;
[...]
}
```

In *packet_received*, poiché la struttura da identificare relativa alle regole multicast è identica a quella relativa alle regole span, si è scelto di aggiungere un controllo, successivamente al parsing di tipo span, sull'indirizzo. Se l'indirizzo di inizio dell'intervallo è di tipo multicast viene eseguito il codice seguente:

```
[...]
if ((first & 0xF000) == 0xF000)
{
  depth = span;
  span = 0x00FF;
  last = first + span;
}
[...]
```

Le informazioni sui gruppi invece vengono trasmesse successivamente alla prima configurazione del nodo sotto forma di pacchetti di tipo *SET* destinati al network layer. L'indirizzo del registro da scrivere corrisponde all'indirizzo di multicast del dominio relativo ai gruppi che si stanno associando al nodo, quindi i primi byte del payload contengono l'indirizzo, mentre i byte successivi contengono i numeri dei gruppi da aggiungere all'elenco delle associazioni (se non viene trasmesso nessun gruppo vengono eliminati tutti i gruppi associati relativi a quel particolare dominio).

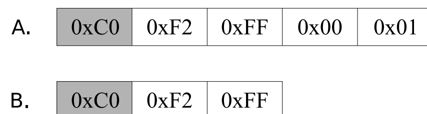


Figura 2.6: *Payload di esempio per la configurazione dei gruppi*

Siano quindi A e B (fig. 2.6) i payload di due pacchetti di tipo *SET*. Quando viene ricevuto il pacchetto A vengono aggiunti i gruppi *00* e *01* per il dominio *2*; invece quando viene ricevuto B vengono eliminati tutti i gruppi associati al dominio *2*.

È stata implementata la funzione `netg_received` puntata da `netogram_received` che viene chiamata alla fine di `packet_received`.

```
int netg_received (uint16_t src_address, uint16_t seq, int8_t code,
↪  uint8_t *data, size_t length) {
  (void)seq;
  (void)src_address;

  if (code != CODE_SET) return -1;

  uint64_t address;
  int8_t offset = from_bin(data,length,&address);

  if (offset<0) return -1;
  if (address>0xFFFF && address<0xF000) return -1;
  uint8_t mc_domain = (uint8_t) (address >> 8) & 0x0F;
  if (offset < length) {
    uint8_t * ptr_end = data + length - 1;
    data = data + offset;
    while (data<=ptr_end) {

      uint8_t mc_group = *(data);
      grp_add(mc_domain, mc_group);
      data++;
    }

  } else if (offset == length) {
    grp_del_dom(mc_domain);
  }

  if (netevent_received) {
    netevent_received(nwk_network_reconf);
  }
  #ifdef S_DEBUG
  grp_dump();
  #endif
  return 0;
}
```

È stata scritta anche la funzione `from_bin` che implementa il parsing degli indirizzi dei registri descritto in [6](pag.29).

Per completezza, anche se non è direttamente collegato al supporto a comunicazioni multicast, è stata anche implementata la memorizzazione nella memoria non volatile della tabella di associazione dei gruppi modificando le funzioni *nwk_save_config* e *nwk_load_config*.

3 Generazione tabelle di routing multicast

Ora che sono definite le regole che determinano il comportamento dei nodi della rete durante l'attraversamento di un pacchetto di tipo multicast, occorre sviluppare una strategia che permetta di calcolare il percorso che un pacchetto deve percorrere tra i nodi della rete affinché tutti i nodi del dominio vengano raggiunti e informati e, da questo, estrarre le tabelle di instradamento da propagare sulla rete in fase di configurazione.

3.1 Principio di creazione dei percorsi

Il problema è simile a quello del commesso viaggiatore noto in teoria dei grafi, dove però il percorso non deve attraversare ogni nodo da informare. Occorre infatti tenere in considerazione che, quando il pacchetto viene trasmesso da un nodo al successivo, questo viene ricevuto da tutti i nodi sufficientemente vicini e connessi allo stesso dominio fisico. Per fare un parallelo con l'esempio del commesso viaggiatore è come se il rappresentante non debba citofonare ad ogni cliente ma debba solo passare per strada notificando della propria presenza tramite un megafono, non dovrà quindi passare davanti ogni porta di ogni strada, ma potrà passare solo lungo alcune strade confidando nella potenza del suo megafono e sull'udito dei possibili clienti. Per quanto possa sembrare faceto e superficiale, questo esempio riporta in maniera efficace tutti i problemi presenti in una architettura del genere.

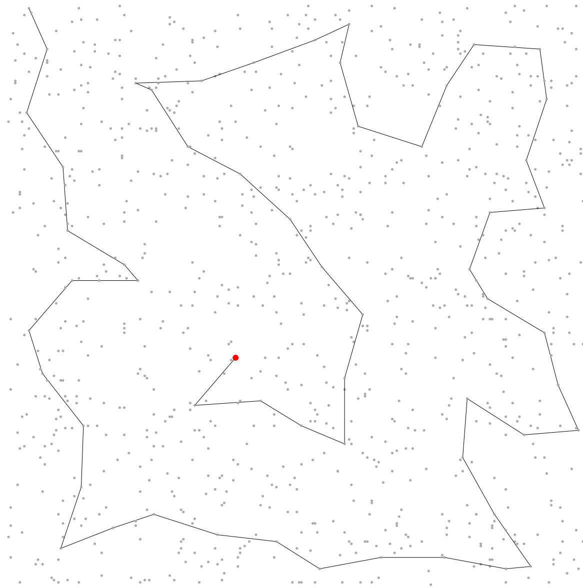


Figura 3.1: *Esempio di un percorso di broadcast*

I percorsi quindi devono originare dal nodo master e propagarsi lungo la rete informando tutti i nodi di un dato dominio di multicast. In fig. 3.1 è riportato un possibile percorso di broadcast. Il nodo master è quello riportato in rosso.

Tutti i nodi della rete (o appartenenti al particolare sottodominio) devono essere raggiunti dal pacchetto che transita lungo il percorso e, allo stesso tempo, il percorso dovrebbe essere composto dal minor numero possibile di salti per fare in modo che la rete sia occupata per il minimo tempo necessario, inoltre ogni trasmissione ha un costo energetico che potrebbe non essere trascurabile nel caso di nodi alimentati a batteria. Quindi l'insieme dei nodi informati dalla ritrasmissione da parte di un nodo appartenente al percorso deve avere una sovrapposizione minima con il sottoinsieme relativo ad un altro nodo del percorso, soprattutto nelle immediate vicinanze del nodo trasmettitore dove la probabilità di trasmissione corretta è alta e una ricezione multipla non porterebbe alcun vantaggio pratico. Se anche un solo nodo del dominio non viene raggiunto il percorso non è da considerarsi accettabile.

In letteratura non esiste un algoritmo noto che risponda esplicitamente alle esigenze di questo problema, però è possibile formularlo come un problema di ricerca di una condizione ottimale. Cioè occorre, a partire da una soluzione sub-ottima non accettabile, per esempio un percorso generato dall'unione di alcuni nodi estratti randomicamente, raggiungere una soluzione ottimale tramite

l'applicazione iterativa di un algoritmo di ottimizzazione.

3.2 Algoritmi genetici

Si è deciso di affrontare il problema utilizzando, per l'ottimizzazione, un algoritmo genetico [7]. Gli algoritmi genetici possono essere utilizzati quando non sono noti altri algoritmi efficienti con complessità lineare o polinomiale. Gli algoritmi genetici sono stati introdotti negli anni '70 da John Holland [8]. Questi sono ispirati ai principi della selezione naturale e dell'evoluzione genetica teorizzati da Charles Darwin nel 1859. Vedremo di seguito una breve descrizione sugli algoritmi genetici in generale e poi una descrizione più approfondita dell'algoritmo implementato per il calcolo dei percorsi di multicast.

Un algoritmo genetico opera su di un insieme di possibili soluzioni del problema chiamato popolazione. Gli individui che compongono la popolazione sono espressione di una certa sequenza di geni. Generalmente questi geni possono assumere valori binari, ovvero determinano la presenza o l'assenza di una certa caratteristica nell'individuo, oppure possono assumere valori reali e quindi determinare l'intensità di una certa caratteristica dell'individuo.

Ad ognuno di questi individui viene associato un punteggio, chiamato valore di fitness, che lo rappresenta sinteticamente: un valore maggiore per la fitness significa che l'individuo è una soluzione migliore del problema. Questo valore può essere calcolato analiticamente valutando una funzione appositamente studiata o può essere estratto da simulazioni atte a valutare le prestazioni dei singoli individui. Applicando delle opportune strategie di incrocio e mutazione agli individui di una popolazione vengono create iterativamente delle nuove popolazioni che vanno a rimpiazzare quelle precedenti. L'algoritmo viene fermato quando vengono soddisfatte le condizioni di arresto.

Quindi, in sintesi, c'è prima una fase di inizializzazione dove viene generata la popolazione iniziale, poi vengono applicate iterativamente le fasi di selezione, incrocio e mutazione fino al soddisfacimento delle condizioni di stop.

3.2.1 Inizializzazione

L'inizializzazione di un algoritmo genetico consiste nella generazione della popolazione iniziale. A seconda della complessità del problema la dimensione

della popolazione può variare ampiamente: dalle poche decine fino a diverse migliaia di possibili soluzioni del problema. Solitamente gli individui della popolazione iniziale sono generati casualmente fino a coprire tutto lo spazio delle soluzioni, ma non è raro che questi vengano generati in zone dove è più probabile trovare soluzioni migliori.

3.2.2 Selezione

Ad ogni generazione una parte della popolazione viene selezionata per la riproduzione. Possono esserci diverse modalità di selezione, ma in ogni caso si cerca di modellare quello che, nella teoria darwiniana, è chiamato "la sopravvivenza del più forte", dove gli individui con un punteggio di fitness più alto hanno una probabilità maggiore di sopravvivere e riprodursi.

3.2.3 Incrocio

Alle coppie estratte dalla procedura di selezione viene applicato un algoritmo di incrocio (crossover). Esistono diverse possibili modalità per effettuare l'incrocio: le strategie di crossover più semplici sono quelle a singolo taglio (fig. 3.2b) e a taglio multiplo (fig. 3.2c) dove vengono segmentati i cromosomi dei genitori in punti casuali e vengono creati uno o due individui discendenti alternando i segmenti del genoma dei due genitori.

Esistono anche delle procedure dove, ogni segmento del genoma dell'individuo discendente viene selezionato da uno dei genitori in maniera stocastica, invece che alternando la provenienza.

3.2.4 Mutazioni

Le mutazioni servono ad evitare, dopo che vengono scartati gli individui con fitness peggiore, che vengano completamente perse delle caratteristiche che potrebbero essere utili nell'esplorazione dello spazio delle soluzioni. Le mutazioni vengono applicate agli individui della popolazione discendente randomicamente con una certa probabilità, solitamente piuttosto bassa. Quando questa avviene si sostituisce un gene con un nuovo valore tra quelli ammissibili.

Queste vanno ad imitare le mutazioni che possono avvenire nella riproduzione di organismi biologici e, come in questi, non c'è alcuna garanzia che l'indivi-

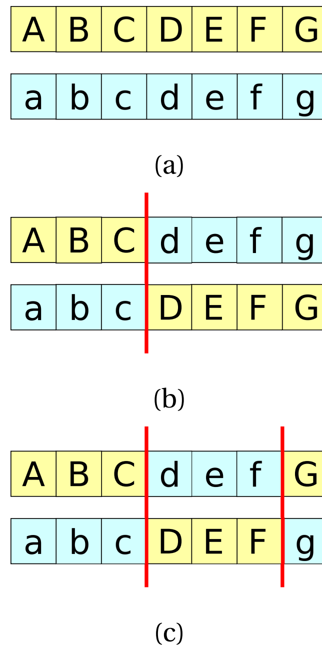


Figura 3.2: *Crossover. a) coppia di individui genitori; b) crossover a singolo taglio; c) crossover a taglio multiplo*

duo risultante sia migliore di quanto non sarebbe stato se la mutazione non fosse occorsa.

Un'altra strategia possibile per introdurre nuove caratteristiche non presenti negli individui della popolazione è la creazione di uno o più nuovi individui.

3.2.5 Interruzione della ricerca

Per come è stato descritto, un algoritmo genetico continuerebbe la generazione di nuove popolazioni indefinitamente, occorre quindi definire delle condizioni che determinano l'interruzione della ricerca della soluzione ottima.

L'algoritmo può venire arrestato quando molti individui della popolazione hanno un valore di fitness vicino a quello dell'individuo migliore, oppure imporre un limite sul numero massimo di generazioni.

3.3 Algoritmo implementato

L'algoritmo che è stato implementato segue in maniera abbastanza fedele lo schema di massima descritto nel paragrafo 3.2. Di seguito verranno descritti tutti

i passi implementati descrivendo, se necessario, il codice sviluppato.

Il calcolo dei percorsi è implementato nella classe *broadcast_handler*. L'inizializzazione e l'esecuzione dell'algoritmo genetico avviene nel costruttore della classe, quindi conviene creare gli oggetti di questa classe (uno per ogni dominio) successivamente al calcolo dell'albero ottimo di unicast e prima della rimozione dei rami di coalbero, in modo da avere ancora il grafo completo di tutte le connessioni fisicamente presenti nella rete.

Verranno descritti di seguito, come per la trattazione generale, i singoli blocchi che compongono l'algoritmo e in seguito si descriverà la logica che determina l'esecuzione dell'algoritmo.

3.3.1 Funzione di fitness

Ogni individuo è un'istanza della classe *individual*, il punteggio di fitness associato all'individuo è salvato nell'attributo *individual::fitness* e viene calcolato nel metodo *individual::calcFitness()*.

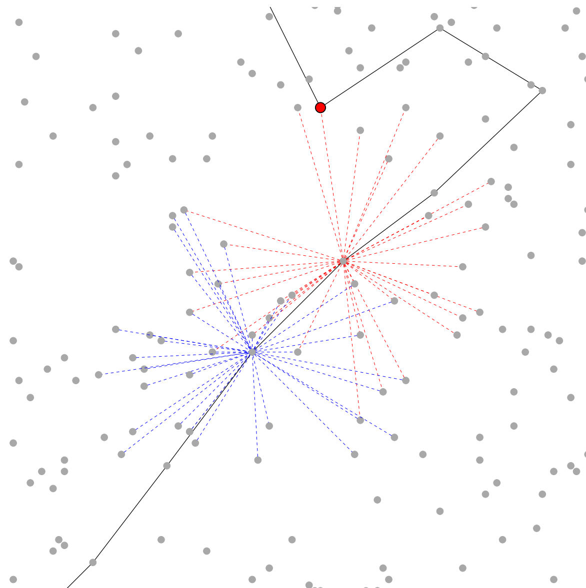


Figura 3.3: *Particolare della rete con in evidenza tutte le connessioni uscenti da due nodi*

L'idea è quella di valutare l'individuo premiando, per ogni nodo del percorso, la presenza di nodi da informare nel range di trasmissione, disincentivando, allo stesso tempo, la presenza di nodi del percorso diversi da quelli immediatamente

precedente e successivo al nodo stesso. Per minimizzare il numero di salti del percorso, viene invece valutata la distanza tra due nodi successivi del percorso penalizzando i collegamenti troppo corti.

Sostanzialmente, il valore della fitness è determinato dalla somma di tre contributi: uno additivo, rappresentante il livello di informazione apportato alla rete in seguito alle ritrasmissioni del pacchetto, che dipende dalle connessioni uscenti presenti in ogni nodo del percorso (archi evidenziati in fig. 3.3). Il secondo, invece, viene sottratto e serve a penalizzare il percorso qualora informasse dei nodi appartenenti al percorso stesso, quindi a penalizzare quei percorsi che passano più volte in una stessa zona e dipende dalle connessioni che congiungono due nodi del percorso. Il terzo è semplicemente il numero di nodi che viene informato dal percorso e serve a fare in modo che, nello spazio delle soluzioni, i massimi relativi della funzione di fitness siano maggiori per quei percorsi che informano tutta la rete.

I valori additivi, come detto, dipendono dal livello di informazione che raggiunge la rete in seguito alla propagazione del pacchetto nella stessa, quindi dipendono direttamente dal reciproco del *Bit Error Rate, o **BER*** (oppure analogamente dal reciproco del *Packet Error Rate, o **PER***).

$$BER_{2\text{-PSK}} = \frac{1}{2}\operatorname{erfc}\left(\sqrt{\frac{E_b}{N_0}}\right) \leq \frac{e^{-\frac{E_b}{N_0}}}{\sqrt{4\pi\frac{E_b}{N_0}}} \tag{3.1}$$

In eq. (3.1) è riportata la formula del BER per trasmissioni binarie e una sua approssimazione accettabile per rapporti segnale/rumore maggiori di 6 dB. Il rapporto E_b/N_0 dipende dal reciproco del quadrato della distanza, infatti:

$$\frac{E_b}{N_0} = \frac{C}{N}\frac{B}{f_b} = \frac{P_{TX}\,G_{TX}\,G_{RX}\,\lambda^2}{(4\pi)^2\,r^2}\frac{B}{N\,f_b} = \frac{K}{r^2} \tag{3.2}$$

dove C, la potenza di portante all'ingresso del ricevitore, è stata stimata con l'equazione di Friis. Quindi, se è noto il BER per una distanza è possibile calcolare il valore della costante K che modella lo scenario propagativo.

Trovato il valore di K è nota una stima del BER al variare della distanza di propagazione. Ovviamente questa stima non è accurata, in quanto non tiene in considerazione l'ambiente reale in cui avviene la propagazione, ma, ai i fini del calcolo della funzione di fitness, può essere sufficiente. Nota la probabilità di errore sul singolo bit si può stimare la probabilità di errore sul pacchetto:

$$PER = 1-(1-BER)^n \tag{3.3}$$

dove n è la lunghezza del pacchetto. Nelle protocollo ToLHnet la lunghezza massima di un pacchetto è di 256 byte (2048 bit).

In fig. 3.4 è riportato il reciproco del PER, stimato come descritto, assumendo un packet error rate pari ad 1 % ad una distanza di 30 m

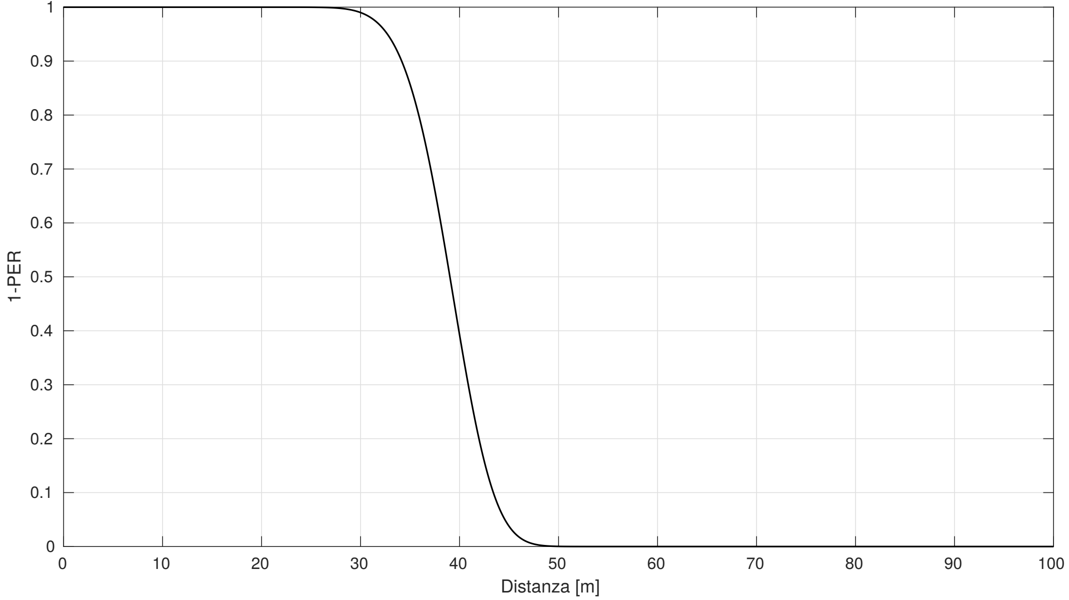


Figura 3.4: *Reciproco del PER*

Per quanto riguarda invece i contributi da sottrarre, sono possibili due casi: se il nodo `N` informa un nodo del percorso, diverso da quello immediatamente precedente (`N-1`) e quello immediatamente successivo (`N+1`), la penalità viene calcolata con la stessa funzione utilizzata per i contributi additivi, in quanto non è importante la distanza del nodo informato, ma piuttosto la sua presenza nel range della trasmissione; se invece si sta considerando la connessione tra il nodo (`N`) e i suoi due vicini più prossimi, poiché si vogliono preferire i percorsi con il minimo numero di salti possibili, quindi quelli con i salti più lunghi, si è utilizzata la funzione gaussiana in fig. 3.5:

$$g(r) = \exp\left(-\frac{[c(r)]^2}{[c(range)]^2}\right) \tag{3.4}$$

con

$$c(r) = cost \cdot \left(1+2\left(\frac{r}{range}\right)^2\right)$$

dove *cost* è il costo base della particolare trasmissione e *range* è la distanza alla quale il PER scende sotto la soglia di accettabilità.

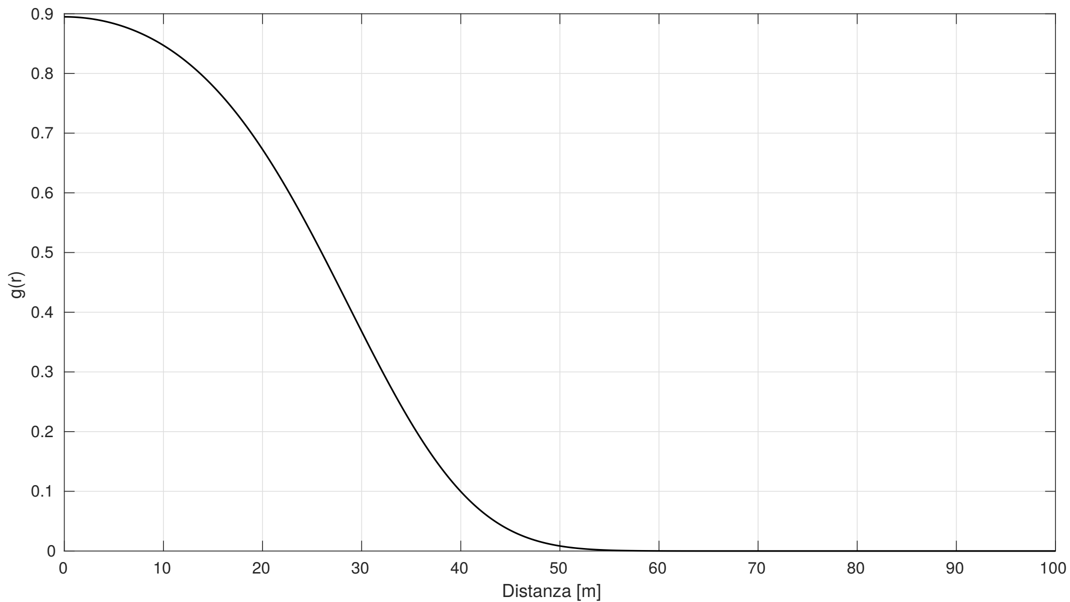


Figura 3.5: *funzione* $g(r)$ *eq.(3.4)*

Riassumendo, quindi, la funzione di fitness è del tipo:

$$F(r) \;=\; \frac{1}{4}\sum_{n=0}^{N}\left(\sum_{m=0}^{M} f(r_{n,m}) - \left[f(r_{n,n-1}) + f(r_{n,n+1})\right] - \sum_{l=0}^{L} g(r_{n,l})\right) + N_{inf} \qquad (3.5)$$

dove N sono i nodi appartenenti al percorso, M sono i nodi direttamente connessi al nodo n, L sono i nodi del percorso direttamente connessi a n e N_{inf} è la quantità di nodi informati. Il fattore $1/4$ serve ad aumentare il peso, nel calcolo del fitness, del numero di nodi informati.

La funzione eq. (3.5) non è propriamente corretta, ma vuole piuttosto essere un riassunto sintetico di quanto descritto, infatti la sommatoria lungo M non è formalmente esatta: se un nodo riceve un pacchetto due volte, la prima con probabilità di ricezione corretta pari a x e la seconda pari a y, non ha senso dire che la probabilità di ricezione corretta sia pari a $x + y$, ma piuttosto questa sarà: $1-(1-x)(1-y)$. Quindi in realtà, più che una somma, è un aggiornamento della probabilità di ricezione corretta condizionata alle ricezioni precedenti.

Nel codice implementato, per semplificare questo aspetto si è deciso di organizzare le varie somme in maniera un po' differente: scorrendo i nodi del percorso vengono aggiornate le probabilità di ricezione corretta associate ai vari nodi

connessi. Questi valori sono memorizzati in un vettore che ha la dimensione dei nodi della rete. Quindi, una volta che è stato scorso tutto il percorso, bisogna totalizzare i valori di questo vettore.

La probabilità che un nodo, dopo aver ricevuto N volte il pacchetto, sia stato informato è pari a:

$$P_N = 1 - \prod_{n=1}^{N} (1 - p_n) \tag{3.6}$$

Quando il pacchetto viene ricevuto un' ulteriore volta, per aggiornare il valore della probabilità, basta calcolare:

$$P_{N+1} = 1 - \prod_{n=1}^{N+1} (1 - p_n) = 1 - \prod_{n=1}^{N} (1 - p_n) \cdot (1 - p_{N+1}) = 1 - (1 - P_N)(1 - p_{N+1}) \tag{3.7}$$

Come ultima cosa vengono conteggiati i nodi unici visitati dal percorso. Se la lunghezza del percorso è maggiore del numero di nodi unici, viene applicata un' ulteriore penalità al fitness per scoraggiare la generazione di percorsi che ripassano su sé stessi.

```
1  void individual::calcFitness()
2  {
3
4    std::set<vertex_descriptor> path_set;
5    fitness = 0;
6    informed_qty = 0;
7    informed = std::vector<bool>(boost::num_vertices(*g), false);
8    bonuses = std::vector<double>(boost::num_vertices(*g), 0);
9    maluses = std::vector<double>(path.size(),0);
10
11   double K=8796.21;
12   unsigned byte_len = 256;
13
14   for (individual_path::iterator it = path.begin(); it != path.end();
      ↪  it++)
15   // scorre il path
16   {
17     path_set.insert(*it);
18
19     boost::graph_traits<graph_t>::out_edge_iterator ei, ei_end;
20     for (boost::tie(ei, ei_end) = boost::out_edges(*it, *g); ei !=
        ↪  ei_end; ++ei)
21     // scorre gli archi connessi ai nodi del path
22     {
23       vertex_descriptor t,s;
24       t = boost::target(*ei, *g);
25       s = boost::source(*ei, *g);
26
```

```
27        double dist = (*pos)[t].dist((*pos)[s]);
28        double PER = pow(1 - erfc(sqrt(K / sqr(dist))) / 2, byte_len *
          ↪  8);
29        // link PER
30
31        bonuses[t] = 1 - ((1 - bonuses[t]) * (1 - PER));
32        // cumulative PER (fitness bonus)
33
34        if (to_inf[t] && bonuses[t] > 0.99)
35        {
36          informed[t] = true;
37        }
38      }
39
40      for (individual_path::iterator vi = path.begin(); vi != path.end();
        ↪  vi++)
41      // scorre il path per cercare se vengono informati nodi del
        ↪  percorso
42      {
43        if (*vi != *(it - 1) && *vi != *(it) && *vi != *(it + 1))
44        {
45          edge_descriptor e;
46          bool b;
47          boost::tie(e, b) = boost::edge(*it, *vi, *g);
48          if (b)
49          {
50            vertex_descriptor t, s;
51            t = boost::target(e, *g);
52            s = boost::source(e, *g);
53            double dist = (*pos)[t].dist((*pos)[s]);
54            double malus = pow(1 - erfc(sqrt(K / sqr(dist))) / 2,
              ↪  byte_len * 8);
55            fitness = fitness - malus;
56            maluses[it - path.begin()] += malus;
57          }
58        }
59        else if (*vi == *(it - 1) || *vi == *(it + 1))
60        {
61          edge_descriptor e;
62          bool b;
63          boost::tie(e, b) = boost::edge(*it, *vi, *g);
64          if (b)
65          {
66            double cost = network.media[(*g)[e].type].cost;
67            double range_cost = cost * 3;
68            double malus = 2 * exp(-sqr((*g)[e].cost) / sqr(range_cost));
69            fitness = fitness - malus;
70            maluses[it - path.begin()] += malus;
71          }
72        }
73      }
74    }
75
76    for (std::vector<bool>::iterator it = informed.begin(); it !=
      ↪  informed.end(); it++)
77    {
78      fitness += bonuses[it - informed.begin()];
```

```
79      if (*it)
80      {
81        informed_qty++;
82      }
83    }
84
85    unique_nodes = path_set.size();
86    fitness=fitness/4;
87    fitness = fitness - (abs(fitness) / 100 * (path.size()-unique_nodes)
     ↪  + informed_qty;
88  }
```

A riga *28* si può vedere il calcolo del packet error rate. Si è scelto di implementare il calcolo utilizzando la funzione errore complementare perché, dopo aver valutato i tempi di esecuzione, questa è risultata essere computazionalmente più veloce sia della funzione approssimata riportata in eq. (3.1) che della stessa implementata sfruttando la funzione *radice quadrata inversa veloce* descritta in [9] e nota per essere stata utilizzata nel motore grafico di *Quake III Arena*. Invece a riga *34* si può vedere che un nodo viene considerato informato se la probabilità di ricezione corretta è maggiore del 99 % e se il nodo appartiene al dominio di multicasting in considerazione.

3.3.2 Generazione della popolazione iniziale

La popolazione iniziale è formata da percorsi che si allontanano radialmente dal nodo master. Questi percorsi non sono difficili da calcolare, basta dividere il piano in settori circolari centrati sul nodo master e cercare il nodo più lontano da esso in ogni settore. In coordinate polari questo si traduce nella semplice ricerca di un massimo. Successivamente basta risalire l'albero ottimo da questi nodi verso il master. Questo è possibile poiché il grafo è già stato pesato dall'algoritmo di Dijkstra per tracciare l'albero di unicast, quindi sono già tracciati i percorsi migliori che congiungono il master a qualsiasi nodo della rete.

Questo è implementato nel metodo
broadcast_handler::generateInitialPopulation()

```
1  void broadcast_handler::generateInitialPopulation(vertex_descriptor
    ↪  &root, std::vector<vertex_descriptor> &parents)
2  {
3    std::vector<unsigned> endp = initialPopulationEndpoints(root);
4
5    for (std::vector<unsigned>::iterator it = endp.begin(); it <
     ↪  endp.end(); it++)
6    {
```

```
7       individual_path path;
8       boost::graph_traits<graph_t>::vertex_descriptor current = *it;
9
10      while (current != root)
11      {
12        path.push_back(current);
13
14        edge_descriptor a;
15        bool b;
16        boost::tie(a, b) = boost::edge(current, parents[current], graph);
17        if (b) graph[a].used = true;
18
19        current = parents[current];
20      }
21      if (path.size() < 3) continue;
22      path.push_back(root);
23
24      individual pop(path, &graph, &poss);
25
26      pop.source=individual::initial;
27      population.push_back(pop);
28    }
29
30    std::sort(population.begin(),population.end(),sortPop);
31  }
32
33  std::vector<unsigned>
     ↪  broadcast_handler::initialPopulationEndpoints(vertex_descriptor
     ↪  &root)
34  {
35    std::vector<unsigned> endp(initial_pop);
36    double a = M_PI * 2 / initial_pop;
37    unsigned sect;
38    for (std::vector<point>::iterator it = poss.begin(); it < poss.end();
       ↪  it++)
39    {
40    sect = floor((*it).getPolar(poss[root]).second / a) +
       ↪  ((double)initial_pop / 2);
41      sect = sect == initial_pop ? 0 : sect;
42      if ((*it).getPolar(poss[root]).first >
         ↪  poss[endp[sect]].getPolar(poss[root]).first)
43      {
44        endp[sect] = it - poss.begin();
45      }
46    }
47    return endp;
48  }
```

La funzione *generateInitialPopulation()* crea gli individui della popolazione iniziale risalendo la struttura *parents* a partire dai nodi estremanti calcolati dalla funzione *initialPopulationEndpoints()*. Come si può vedere questa divide l'angolo giro in *initial_pop* settori di pari ampiezza e, scorrendo tutti i nodi della rete, cerca i più distanti dal nodo master in ogni settore.

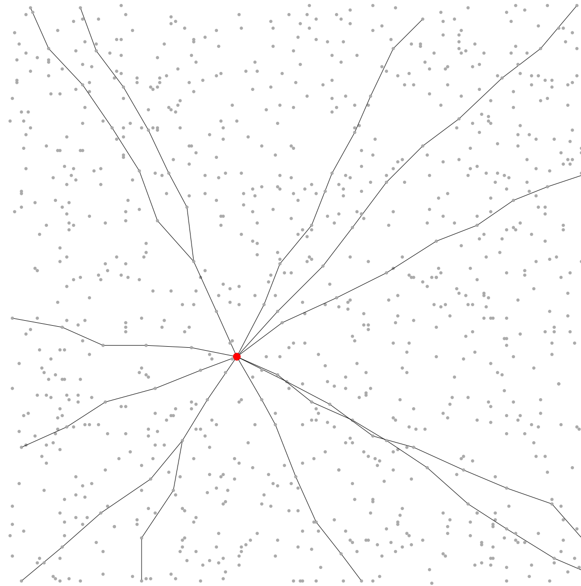


Figura 3.6: *Popolazione iniziale*

Gli individui trovati vengono scartati se sono composti da meno di tre nodi, questo perché sarebbero inutili nel generare nuovi individui in quanto non presentano punti validi dove applicare il crossover.

3.3.3 Selezione

Sono stati esaminati due metodi di selezione dei candidati per il crossover. Il primo consiste nel far incrociare i due individui con il punteggio di fitness maggiore con tutti gli altri; l'altro consiste nell'estrazione di coppie casuali fino al raggiungimento della quantità imposta per gli individui della discendenza.

Il secondo metodo è risultato migliore poiché i geni potenzialmente utili degli individui peggiori hanno più probabilità di venire combinati in maniera efficace.

Poiché, per problemi di compatibilità con le revisioni più recenti del preprocessore C++, non è possibile compilare il codice utilizzando le versioni più recenti del C++, l'estrazione di un elemento random dalla popolazione è stato implementato tramite la funzione *rand()*:

```
[...]
  unsigned rand1, rand2;
  rand1 = rand() % population.size();
  do
  {
```

```
      rand2 = rand() % population.size();
    } while (rand2 == rand1);

    crossover(population[rand1], population[rand2]);
  [...]
```

Un possibile problema di questo metodo è che i valori estratti potrebbero non avere una distribuzione esattamente uniforme: la funzione *rand()* restituisce un intero con probabilità uniformemente distribuita compreso tra 0 e *RAND_MAX* (che dipende dall'architettura e dal compilatore, per il compilatore GCC solitamente è pari al massimo intero rappresentabile, quindi per una macchina a 32bit $2^{32} - 1$, ma non è raro incontrare compilatori per cui questo valore è pari a *0x7FFF*) quindi ogni elemento ha probabilità di essere estratto pari a 1/*RAND_MAX*, ma, nel momento in cui viene applicata la funzione modulo, se la ragione del modulo non è un divisore di *RAND_MAX* alcuni valori avranno una probabilità maggiore di altri di essere estratti. Questo scarto di probabilità tra gli esiti è pari a 1/*RAND_MAX*, quindi, ai fini dell'estrazione di individui della popolazione, questa non uniformità può essere considerata trascurabile.

Si veda il seguente esempio per comprendere quanto appena affermato. Si supponga di disporre di un processo stocastico che generi i valori compresi tra 0 e 9 ognuno con una probabilità pari a 1/10. Come si può vedere in fig. 3.7, applicando la funzione resto per restringere l'intervallo della variabile aleatoria ai valori compresi tra 0 e 2 lo 0 ha una probabilità di essere estratto pari 4/10 mentre gli altri di 3/10.

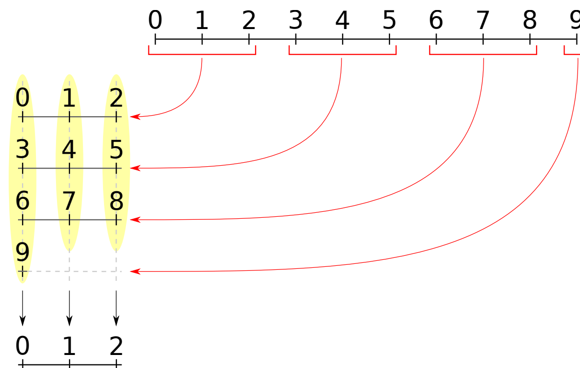


Figura 3.7: *Operazione modulo*

3.3.4 Crossover

Il crossover implementato è a singolo punto di taglio (fig. 3.8). Si è scelto quindi di non implementare il meccanismo che determina in maniera randomica il genitore da cui ereditare le singole porzioni di genoma, poiché altrimenti ci sarebbe una probabilità troppo alta (2 casi su 4) che gli individui discendenti siano identici ai genitori. Questo porterebbe ad uno spreco computazionale perché la procedura di crossover genererebbe al 50 % dei casi degli individui che sarebbero quasi certamente scartati. Quindi, una volta estratti i nodi degli individui genitori dove avverranno i tagli per il crossover (si vedranno più avanti le modalità con cui avviene questa selezione), si procederà a creare gli individui discendenti. Poiché i geni di un individuo sono i nodi che il percorso attraversa, creando un nuovo individuo da segmenti di individui distinti significherebbe, con buona probabilità, creare un percorso formato da tratti disgiunti (fig. 3.8b). Occorre quindi calcolare dei segmenti che raccordino gli estremi di questi tratti disgiunti (fig. 3.8c).

Una prima idea per il calcolo di queste giunzioni potrebbe essere l'utilizzo dell'algoritmo di Dijkstra. Questo garantisce di trovare sempre un percorso tra due nodi (ammesso che il grafo sia connesso), ma questo significherebbe ricalcolare ogni volta il potenziale dell'intero grafo anche se si vogliono congiungere due nodi molti vicini. Si è deciso allora di applicare una ricerca non ottima basata sulla posizione dei punti: supponendo di voler trovare un percorso che unisca un nodo `A` ad un nodo *B*, viene cercato nell'insieme formato dal nodo `A` e dai nodi adiacenti ad esso, il nodo `A'` più vicino a *B*. Sarà ora sufficiente reiterare la ricerca considerando il nodo `A'` come nuovo punto di partenza.

Se il nodo `A'` trovato coincide con `B` allora la ricerca è terminata, se invece coincide con `A` significa che la ricerca è bloccata e non può proseguire. In questo caso viene scartata la prima parte di percorso trovata e si applica la ricerca computazionalmente più costosa usando l'algoritmo di Dijkstra. Questa ricerca è implementata nel metodo *`broadcast_handler::createJunction()`*.

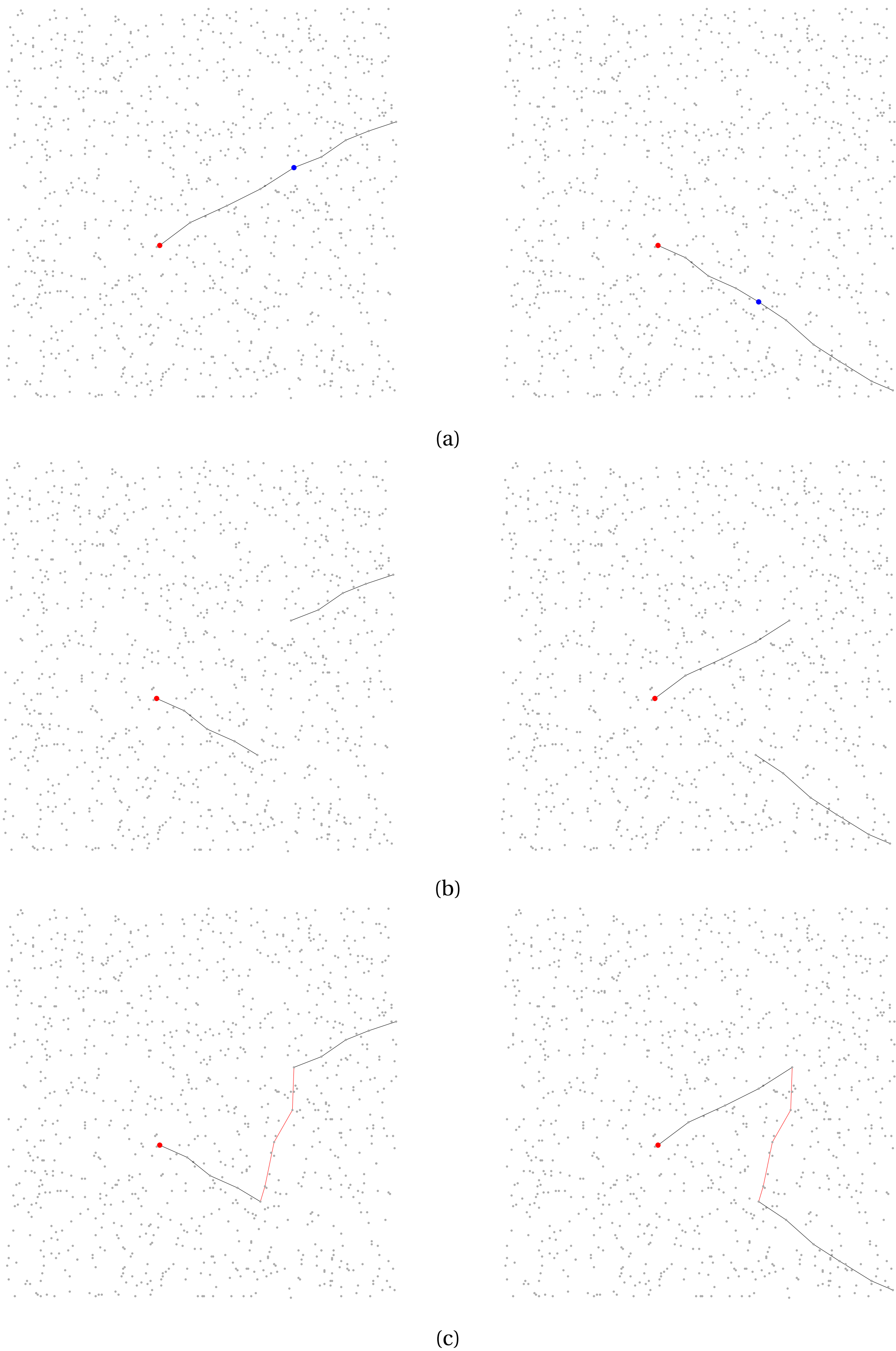


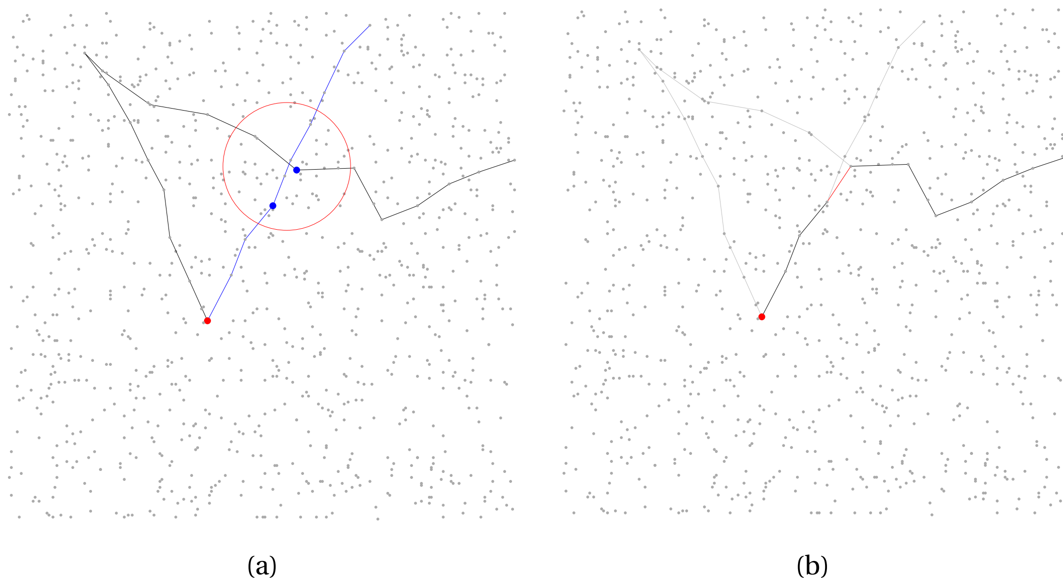
Figura 3.8: *Crossover. a) coppia di individui genitori; b) segmenti che formano gli individui discendenti; c) individui discendenti completi*

```
1  std::vector<vertex_descriptor>
    ↪  broadcast_handler::createJunction(vertex_descriptor a,
    ↪  vertex_descriptor b)
2  {
3    std::vector<vertex_descriptor> junction;
4    junction.push_back(a);
5
6    while (*(junction.end() - 1) != b)
7    {
8      junction.push_back(findNextNode(*(junction.end() - 1), b));
9
10     if ((*(junction.end() - 1) == (*(junction.end() - 2))))
11     // se findNextNode ha restituito due volte lo stesso nodo
12     {
13       boost::property_map<graph_t, double edge_property::*>::type
          ↪  weights = boost::get(&edge_property::cost, graph);
14       std::vector<vertex_descriptor>
          ↪  parents(boost::num_vertices(graph));
15       boost::dijkstra_shortest_paths(graph, b,
          ↪  boost::weight_map(weights).predecessor_map(&parents[0]));
16
17       std::vector<vertex_descriptor> path;
18       boost::graph_traits<graph_t>::vertex_descriptor current = a;
19
20       while (current != b)
21       {
22         path.push_back(current);
23         current = parents[current];
24       }
25       path.push_back(b);
26       return path;
27     }
28   }
29   return junction;
30 }
31
32
33 vertex_descriptor broadcast_handler::findNextNode(vertex_descriptor a,
    ↪  vertex_descriptor b)
34 {
35   vertex_descriptor next = a;
36   boost::graph_traits<graph_t>::adjacency_iterator vit, vend;
37   boost::tie(vit, vend) = boost::adjacent_vertices(a, graph);
38
39   while (vit != vend)
40   {
41     double range = network.media[graph[(boost::edge(a, *vit,
        ↪  graph).first)].type].range;
42
43     if (poss[*vit].dist(poss[a]) < range && poss[*vit].dist(poss[b]) <
        ↪  poss[next].dist(poss[b]))
44     {
45       next = *vit;
46     }
47     vit++;
48   }
```

```
49
50     return next;
51 }
```

Il metodo *broadcast_handler::findNextNode(...)* chiamato a riga *8* restituisce il nodo più vicino a *b* tra quelli adiacenti all'ultimo nodo del vettore *junction*.

Per quanto riguarda la selezione dei nodi dove avviene il taglio per il crossover sono state implementate due soluzioni. La prima (fig. 3.8a) è un'estrazione casuale dove ogni nodo, ad esclusione del master, ha la stessa probabilità di essere estratto: questo sistema è valido per la prima parte della ricerca quando i percorsi non si sviluppano su tutta la superficie della rete e quindi unire due percorsi aumenta la superficie esplorata. Nel secondo metodo (fig. 3.9a), invece, la probabilità di estrazione non è uniforme in quanto si vuole che i punti di crossover siano nelle regioni dove gli individui estratti per il crossover passino vicini tra loro.



(a) (b)

Figura 3.9: *Esempio seconda strategia. a) coppia di individui genitori; b) individuo discendente*

Come si può vedere in fig. 3.9b i punti di taglio sono stati estratti in prossimità dell'intersezione tra i due percorsi, quindi nella zona dove la probabilità è più alta. In questo modo le giunzioni generate saranno, con alta probabilità, corte. Questa strategia è migliore nelle fasi più avanzate della ricerca, quando i percorsi

esplorano già gran parte della rete e una giunzione tra due punti casuali porterebbe ad attraversare aree già esplorate e quindi ad una penalità sul punteggio di fitness.

Come per la prima strategia riportata in fig. 3.2, viene generato anche il percorso formato dai segmenti complementari che qui, per brevità, non è stato riportato.

```
1  void broadcast_handler::crossover2(individual &a, individual &b)
2  {
3    individual_path o1, o2;
4
5    std::vector<double> cdfa;
6    cdfa = cdfBranching(a,b);
7    unsigned cut_a=0;
8    double randn;
9
10   do
11   {
12     randn = (double)rand() / RAND_MAX;
13     for (std::vector<double>::iterator it = cdfa.begin(); it !=
       ↪ cdfa.end(); it++)
14     {
15       if (*it < randn)
16       {
17         cut_a = it - cdfa.begin() + 1;
18       }
19       else
20       {
21         break;
22       }
23     }
24   } while (cut_a == (cdfa.size()));
25
26   // cerco il nodo di b più vicino a cut_a
27   unsigned cut_b = 1;
28   for (individual_path::iterator path_it = b.path.begin() + 1; path_it
     ↪ != b.path.end() - 1; path_it++)
29   {
30     if (poss[a.path[cut_a]].dist(poss[*path_it]) <
       ↪ poss[a.path[cut_a]].dist(poss[b.path[cut_b]]))
31     {
32       cut_b = path_it - b.path.begin();
33     }
34   }
35
36 [...] // concatenazione segmenti
37
38   oo1.source=individual::cross2;
39   oo2.source=individual::cross2;
40
41   if (oo1.fitness > oo2.fitness)
42   {
43     offspring.push_back(oo1);
```

```
44      }
45      else
46      {
47        offspring.push_back(oo2);
48      }
49      return;
50    }
```

La funzione *cdfBranching()* chiamata a riga *5* restituisce il vettore di probabilità cumulativa relativa ai punti di taglio del percorso *a* associata al percorso *b*. Nel ciclo *do while* a riga *10* viene estratto il punto di taglio del primo percorso. Il punto di taglio del secondo percorso è il nodo più vicino al punto estratto del primo percorso.

```
1   std::vector<double> cdfBranching(individual &a, individual &b)
2   {
3     std::vector<double> v;
4     for (individual_path::iterator it = a.path.begin(); it !=
     ↪  a.path.end(); it++)
5     {
6       v.push_back(b.bonuses[*it]);
7     }
8     for (std::vector<double>::iterator it = v.begin() + 1; it != v.end();
     ↪  it++)
9     {
10      *it += *(it - 1);
11    }
12    for (std::vector<double>::iterator it = v.begin(); it != v.end();
     ↪  it++)
13    {
14      *it = *(it) / *(v.end() - 1);
15    }
16    return v;
17  }
```

Per calcolare il vettore di probabilità cumulativa viene costruito il vettore *v* prendendo gli elementi del vettore *bonuses* del secondo percorso corrispondenti ai nodi attraversati dal primo percorso. Si calcolano poi le somme parziali e si normalizza rispetto al valore massimo. Questo viene descritto più in dettaglio in appendice A.

3.3.5 Mutazioni

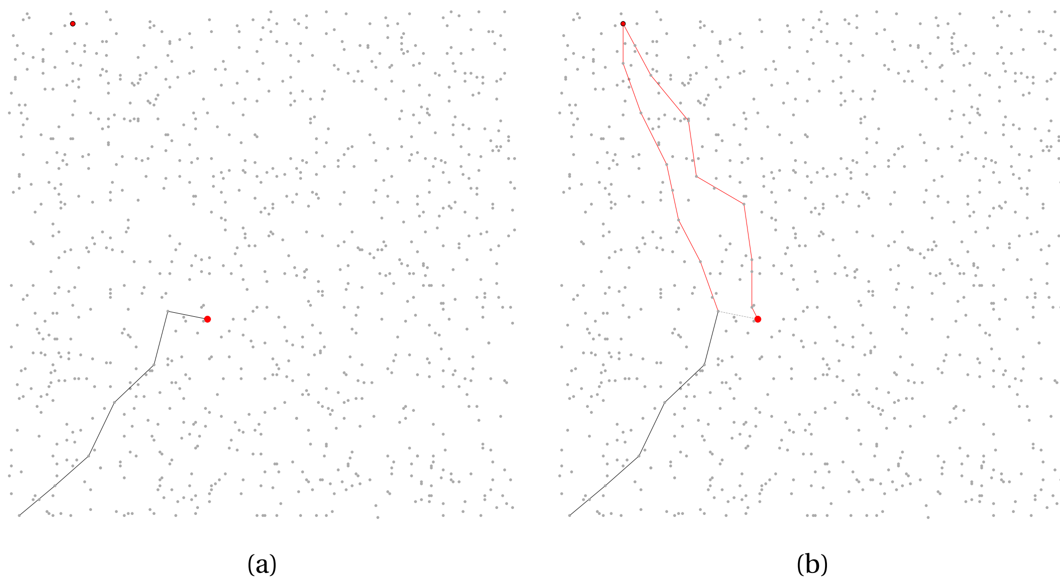
Una volta che la fase di crossover è terminata e l'insieme di individui discendenti ha raggiunto la dimensione imposta vengono generati gli individui mutanti con una probabilità iniziale del 5 %.

Sono stati implementati due meccanismi di mutazione, il primo viene applicato nel caso in cui l'individuo sul quale viene applicata la mutazione non informi tutta la rete e serve ad aumentare il numero di nodi raggiunti, il secondo invece nel caso in cui il percorso informi tutta la rete e serve ad aumentare il punteggio di fitness riducendo le penalità associate al percorso.

Mutazione 1

Viene estratto, con probabilità uniforme, un nodo `N` random dall'insieme dei nodi non informati e si cerca il nodo `M` del percorso più vicino a `N`. Viene creato quindi il nuovo individuo formato da:

- il tratto del percorso originale tra il nodo iniziale e il nodo `M-1`;
- la giunzione tra `M-1` e `N`;
- la giunzione tra `N` e `M`;
- il tratto del percorso originale tra il nodo `M` e il nodo finale.



(a) (b)

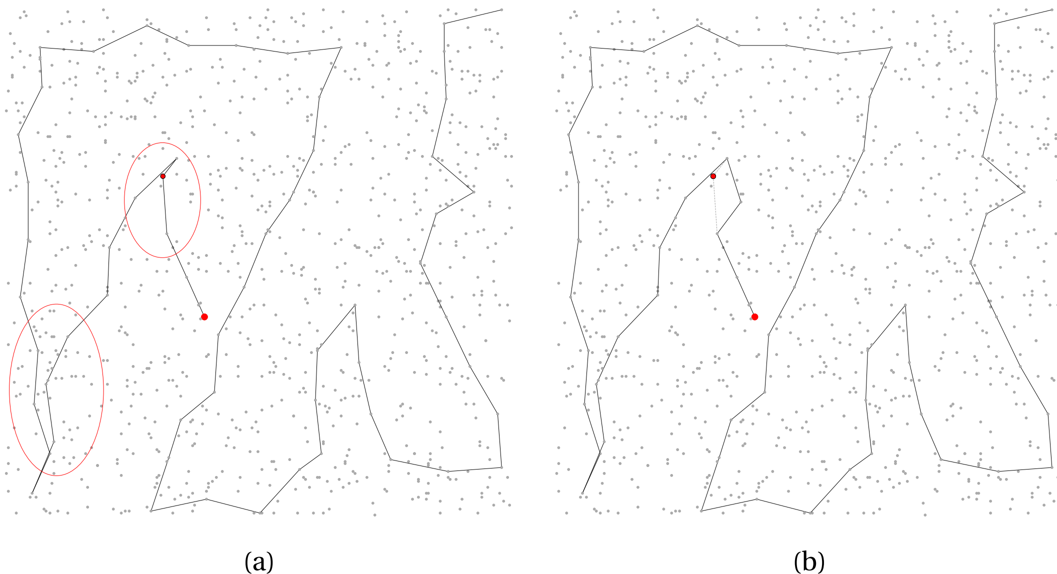
Figura 3.10: *Esempio di mutazione 1. a) individuo di partenza con evidenziato il nodo estratto; b) individuo mutato*

Mutazione 2

Viene estratto un nodo `N` del percorso con una funzione di probabilità legata ai malus sul fitness associati ai vari nodi. Viene creato quindi il nuovo individuo formato da:

- il tratto del percorso originale tra il nodo iniziale e il nodo `N-a`;
- la giunzione tra `N-a` e `N+b`;
- il tratto del percorso originale tra il nodo `N+b` e il nodo finale.

`a` e `b` inizialmente sono fissati entrambi a 1 e, se `N-a` e `N+b` sono nodi che non devono essere informati, vengono incrementati finché non vengono trovati due nodi appartenenti al dominio. Quindi, se i nodi `N-1` e `N+1` appartengono al dominio viene eliminato dal percorso solo il nodo `N`, altrimenti viene eliminato il segmento di lunghezza massima che contiene `N` formato per intero da nodi non appartenenti al dominio. Se il segmento che viene eliminato è all'inizio del percorso viene ovviamente mantenuto il nodo master da cui partono tutti i percorsi, se invece è alla fine viene troncato il percorso.



(a) (b)

Figura 3.11: *Esempio di mutazione 2. a) individuo di partenza con evidenziate le zone con probabilità maggiore e il nodo estratto; b) individuo mutato*

Come si può vedere dall'esempio in fig. 3.11 il nodo estratto, evidenziato in rosso, è particolarmente vicino ad altri nodi appartenenti al percorso, quindi, tra i contributi che formano il valore di fitness, ci sarà una penalità non nulla dovuta al nodo estratto. Ci si aspetterebbe quindi che eliminandolo dal percorso il punteggio di fitness aumenti. Esiste, tuttavia, una probabilità non nulla che, una volta ricongiunti i nodi `N-a` e `N+b`, l'individuo mutato sia identico a quello di partenza. In tal caso verrà successivamente scartato come duplicato.

Per creare le giunzioni viene usata la funzione *`createJunction()`* già descritta in precedenza, mentre per l'estrazione del punto di mutazione per il secondo algoritmo vengono applicate procedure analoghe a quelle descritte nella sezione 3.3.4.

```
1   individual broadcast_handler::mutation1_ret(individual &pop)
2 {
3   std::vector<double> cdf = cdfMut(pop);
4   unsigned cut = 0;
5
6   do {
7     cut = 0;
8     double randn = (double)rand() / RAND_MAX;
9     while (randn > cdf[cut]) cut++;
10  } while (cut == pop.path.size() - 1);
11
12  individual_path newpath;
13  unsigned a=1,b=1;
14
15  while ((int)(cut - a) != -1 && !to_inf[pop.path[cut - a]])
16  {
17    a++;
18  }
19  while (!to_inf[pop.path[cut + b]] && cut + b != pop.path.size()-1)
20  {
21    b++;
22  }
23
24  if ((int)(cut - a) != -1)
25  {
26    newpath.insert(newpath.end(), pop.path.begin(), pop.path.begin() +
       ↪  cut - a);
27    individual_path junct = createJunction(*(pop.path.begin() + cut -
       ↪  a), *(pop.path.begin() + cut + b));
28    newpath.insert(newpath.end(), junct.begin(), junct.end() - 1);
29  }
30  newpath.insert(newpath.end(), pop.path.begin() + cut + b,
     ↪  pop.path.end());
31
32  individual new_pop(newpath, pop.g, pop.pos, to_inf);
33  new_pop.source=pop.source;
34  return new_pop;
35 }
```

3.3.6 Struttura dell'algoritmo

Ora che sono stati descritti i blocchi fondamentali che costituiscono l'algoritmo è possibile descrivere la sequenza di operazioni che formano la fase di inizializzazione e il ciclo di generazione.

```
1  broadcast_handler::broadcast_handler(std::string name, graph_t &g,
    ↪ std::deque<node_description> &nodes, vertex_descriptor &root,
    ↪ std::vector<vertex_descriptor> &parents, std::vector<bool>
    ↪ to_inf):name(name), to_inf(to_inf)
2  {
3    nodes_m = &nodes;
4    graph = g; // salvo il grafo completo
5
6
7    unsigned num_nodes = 0;
8    for (unsigned i = 0; i < to_inf.size(); i++)
9    {
10     if (to_inf[i]) num_nodes++;
11   }
12
13   boost::graph_traits<graph_t>::edge_iterator ei, ei_end;
14   for (boost::tie(ei, ei_end) = boost::edges(graph); ei != ei_end;)
15   graph[(*ei++)].used = false;
16
17   srand(time(0));
18
19   parse_conf("broadcast.txt");
20
21   for (unsigned vi = 0; vi < boost::num_vertices(graph); vi++)
22   {
23     node_description const &ni = nodes[vi];
24     node_description::connection ci = {'\0'};
25     uint16_t di = 0;
26     boost::graph_traits<graph_t>::out_edge_iterator ei, ei_end;
27     for (boost::tie(ei, ei_end) = boost::out_edges(vi, graph); ei !=
        ↪ ei_end; ++ei)
28     {
29       if (boost::target(*ei, graph) == parents[vi])
30       {
31         di = graph[*ei].domain;
32         break;
33       }
34     }
35     for (unsigned l = 0; l < ni.num_connections; ++l)
36     if (ni.connections[l].domain == di)
37     {
38       ci = ni.connections[l];
39     }
40     point p(ci.pos_x,ci.pos_y);
41     poss.push_back(p);
42   }
```

Per prima cosa vengono conteggiati i nodi che devono essere informati, righe `7-11`, poi viene inizializzato il generatore di sequenze pseudo-casuali utilizzando

come seme l'orario di sistema in modo da avere una condizione iniziale diversa ogni volta che viene eseguita la ricerca. Alla riga *19* viene caricato il file di configurazione per la generazione dei percorsi di multicast. In questo file sono definiti il numero di individui della popolazione iniziale, gli individui massimi per la popolazione e per i discendenti e la probabilità iniziale di mutazione. Nel ciclo *for* a riga *21* viene riempito il vettore *poss* che contiene le coordinate dei nodi della rete. Queste coordinate sono salvate in oggetti di classe *point*

```
class point
{
  public:
  double x, y;
  point(double x = 0, double y = 0);
  std::pair<double, double> getCart();
  std::pair<double, double> getPolar(double x0 = 0, double y0 = 0);
  std::pair<double, double> getPolar(point p0);
  double dist(double x0 = 0, double y0 = 0);
  double dist(point p0);

  point operator+(point &);
  point operator-(point &);
};
```

I metodi *point::getPolar(...)* servono ad ottenere la posizione del punto in coordinate polari attorno al polo *p0*, questo è utile nella generazione della popolazione iniziale. I metodi *point::dist(...)* restituiscono la distanza del punto da *p0*.

```
43    generateInitialPopulation(root, parents);
44
45  [...]
46
47    unsigned i=0;
48    std::vector<individual> good_pop;
49    while (population[0].informed_qty!=num_nodes)
50    {
51      generation();
52
53      std::cout << "[A" << ++i << "] ";
54      population[0].print();
55
56      for (std::vector<individual>::iterator it = population.begin(); it
       ↪  < population.end(); it++)
57      {
58        if (it->informed_qty == num_nodes)
59        {
60          good_pop.push_back(*it);
61        }
62      }
```

```
63
64      std::sort(good_pop.begin(), good_pop.end(), sortPop);
65      std::vector<individual>::iterator last =
         ↪  std::unique(good_pop.begin(), good_pop.end());
66      last = last - good_pop.begin() > 50 ? good_pop.begin() + 50 : last;
67      good_pop.erase(last, good_pop.end());
68
69      if (i > 80 && good_pop.size() > 0 && good_pop[0].fitness /
         ↪  population[0].fitness > 0.9)
70      {
71        break;
72      }
73
74    }
75
76  [...]
77  }
```

Dopo aver riempito il vettore delle posizioni si procede con la generazione della popolazione iniziale e si avvia quindi la prima fase di ricerca. Questo primo ciclo di evoluzione si interromperà quando il miglior individuo di una generazione informerà tutti i nodi che devono essere informati (l'intera rete per il percorso di broadcast o un sottoinsieme per quelli di multicast). In questo ciclo viene semplicemente chiamato il metodo *broadcast_handler::generation()* e, una volta terminata la generazione, vengono stampate a video le caratteristiche dell'individuo migliore, per esempio:

[A5] Path len: 67 jumps; Unique nodes visited: 65; Fitness: 969.164; Informed nodes: 845

A5 indica che è stata completata la quinta generazione della prima fase, *Path len* è la quantità di nodi attraversati dal percorso, *Unique nodes visited* è il numero di nodi attraversati contati una volta sola. Se il primo valore è maggiore del secondo vuol dire che il percorso attraversa qualche nodo più di una volta. Vengono inoltre inseriti nel vettore *good_pop* tutti gli individui di *population* che informano tutti i nodi che devono essere informati, questo è utile nel caso in cui non si arrivi alla condizione di arresto e si debba quindi interrompere l'evoluzione, il vettore conterrà comunque dei percorsi che possono essere utilizzati per informare la rete.

Come ultimo passo della generazione viene scartata la popolazione attuale e i primi *max_population* individui del vettore *offspring* diventano la nuova popolazione per la successiva generazione.

La prima fase di evoluzione viene interrotta, se non si arriva alla situazione che l'individuo migliore informa tutta la rete, non prima di 80 iterazioni e solo se l'individuo migliore di *good_pop* abbia un punteggio di fitness di almeno il 90 % del migliore individuo di *population*, righe *69-72*. Una volta che viene completata la prima fase di evoluzione, viene portata la probabilità di mutazione a 80 % e viene eseguito un altro ciclo evolutivo identico al primo che si interrompe dopo 30 iterazioni. Questo viene fatto nella speranza che le mutazioni migliorino ulteriormente gli individui. Ultimato anche questo ciclo, l'individuo migliore sarà il primo del vettore *good_pop*.

Nel metodo *broadcast_handler::generation()* vengono quindi applicate le varie strategie di crossover e mutazione. Per prima cosa vengono salvati i 5 individui migliori della generazione corrente e poi si applica il crossover. Come visto in section 3.3.4 sono state previste due possibili strategie di crossover, occorre quindi decidere, per ogni generazione, quale strategia applicare. Per mantenere un comportamento stocastico, tipico degli algoritmi genetici, si è deciso di applicare una strategia casuale per ogni coppia di genitori con una probabilità che venga applicata l'una piuttosto che l'altra legata ai punteggi di fitness degli individui della popolazione attuale. Nell'attributo *individual::source* è annotata, sotto forma di variabile enum, l'origine del particolare individuo:

- *initial*: l'individuo è uno della popolazione iniziale
- *prev*: l'individuo è uno dei *best fit* della generazione precedente
- *cross1*: l'individuo è stato generato con la prima tipologia di crossover
- *cross2*: l'individuo è stato generato con la seconda tipologia di crossover

Vengono calcolate quindi le quattro medie della fitness per gli individui che hanno la stessa origine. Per la prima generazione l'unica media non nulla sarà, ovviamente, quella relativa agli individui della popolazione iniziale quindi le due strategie avranno la stessa probabilità. Altrimenti avrà una probabilità maggiore la strategia che, nella generazione precedente, ha generato individui migliori.

```
[...]
  double th = .5;

  if (qty[individual::cross1] && qty[individual::cross2])
  {
```

```
    th = medie[individual::cross1] < medie[individual::cross2] ? 0.2 :
     ↪  0.8;
  }

  while (offspring.size() < max_offspring)
  {
    unsigned rand1, rand2;
    rand1 = rand() % population.size();
    do {
      rand2 = rand() % population.size();
    } while (rand2 == rand1);

    if ((double)rand() / RAND_MAX < th)
    {
      crossover(population[rand1], population[rand2]);
    } else {
      crossover2(population[rand1], population[rand2]);
    }
  }
[...]
```

Come si può vedere quindi, una volta stabilite le probabilità con cui avvengono le due tipologie di crossover, vengono estratte coppie casuali di individui per l'incrocio finché il vettore *offspring*, che contiene gli individui discendenti, non è della dimensione massima specificata.

Successivamente si calcolano le mutazioni:

```
[...]
  std::vector<individual> mutants;
  for (std::vector<individual>::iterator it = offspring.begin(); it !=
   ↪  offspring.end(); it++)
  {
    if (rand() < mutation_chance * RAND_MAX)
    {

      mutants.push_back(mutation_ret(*it));
    }
  }
[...]
```

dentro *broadcast_handler::mutation_ret(individual)* viene controllato se l'insieme dei nodi non informati è vuoto e, nel caso, applica l'altra mutazione:

```
individual broadcast_handler::mutation_ret(individual & pop)
{
  std::vector<vertex_descriptor> uninformed;
  for (std::vector<bool>::iterator inf_it = pop.informed.begin();
   ↪  inf_it != pop.informed.end(); inf_it++)
  {
    if (!*inf_it) uninformed.push_back(inf_it - pop.informed.begin());
  }
```

```
    if (uninformed.size()==0){
      return mutation1_ret(pop);
    }
  [...]
```

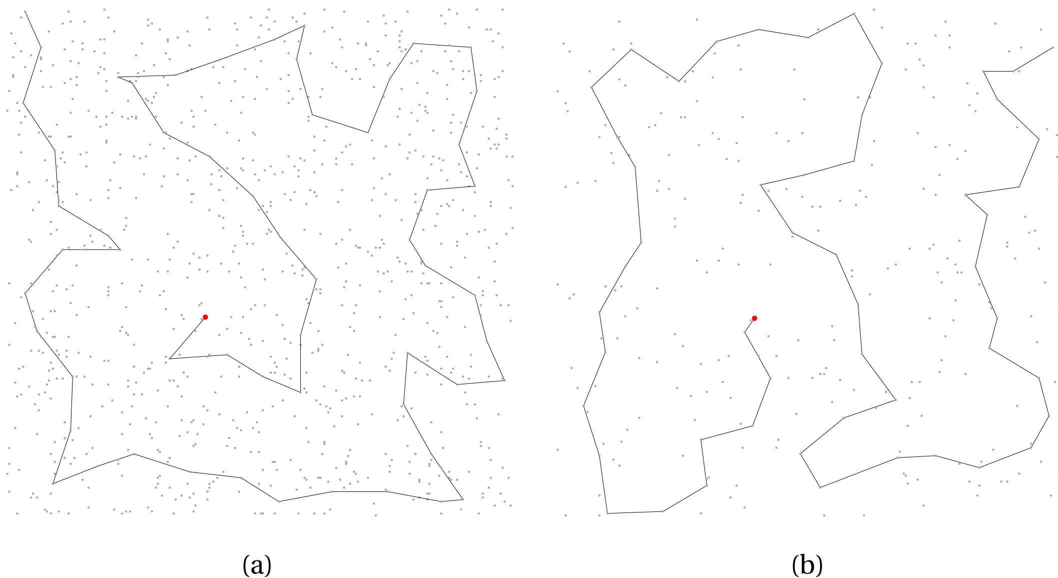
Ultimato il calcolo degli individui mutanti, gli individui di *offspring* vengono ordinati per fitness. Se l'individuo migliore di *offspring* è lo stesso di *population*, poiché il crossover non è stato efficace nel generare individui migliori, viene aumentata la probabilità di mutazione dello 0,5 %.

4 Analisi dei risultati

Si andranno ora ad analizzare alcuni percorsi ottenuti tramite l'algoritmo genetico per valutarne l'efficacia nell'estrazione di percorsi utili. Verranno fatte poi anche delle considerazioni sulla perdita dei pacchetti che si avrebbe utilizzando questi percorsi.

4.1 Analisi della generazione dei percorsi

Sono stati generati più percorsi considerando diverse distribuzioni di nodi e diversi domini di multicast.



(a) (b)

Figura 4.1: *a) rete quadrata con 1000 nodi; b) rete quadrata con 300 nodi*

In fig. 4.1 sono riportate due reti quadrate, entrambe di 256 m di lato, la prima contenente 1000 nodi e la seconda 300. Come si può vedere in entrambi i casi l'algoritmo ha generato un percorso valido. Nel secondo caso l'algoritmo ha richiesto una quantità maggiore di iterazioni poiché, essendo minore la densità dei nodi, è meno probabile che un percorso che informi la totalità dei nodi sia anche quello con la fitness più alta. Per raggiungere l'interruzione dell'evoluzione si è dovuto attendere il soddisfacimento delle condizioni secondarie di arresto, ma poiché la rete è composta da molti meno nodi, i tempi di esecuzione nei due casi sono risultati paragonabili. La posizione del nodo master, per una rete simmetrica come queste, non influenza le prestazioni dell'algoritmo.

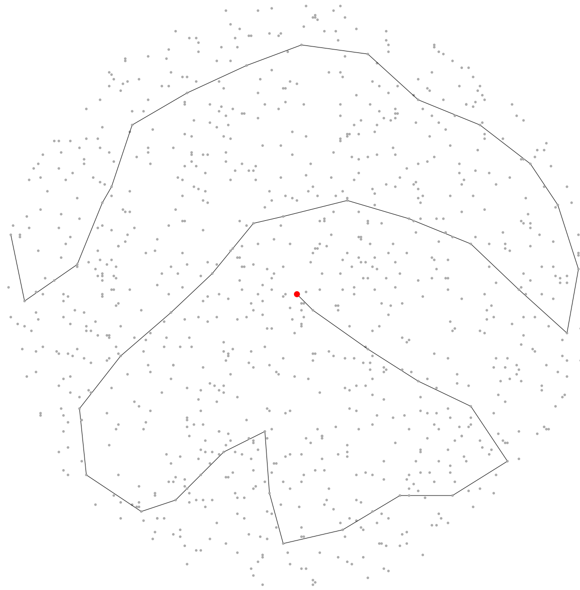


Figura 4.2: *Rete circolare con 1000 nodi*

In fig. 4.2 è riportata una rete circolare di diametro pari a 256 m contenente 1000 nodi con il master è stato posizionato nel centro. Anche in questo caso il percorso estratto appare valido e simile a uno che si sarebbe potuto tracciare manualmente. Anche in questo caso la posizione del master non altera significativamente le prestazioni dell'algoritmo.

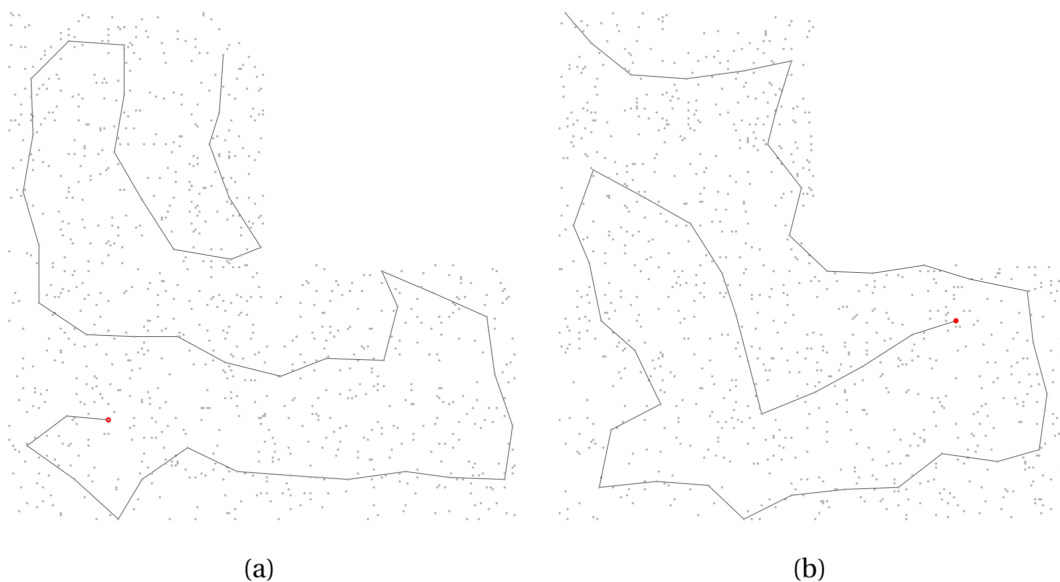


Figura 4.3: *Rete poligonale con diverse posizioni del nodo master*

In questi ultimi due casi la rete non varia, è composta da 1000 nodi ed è inscritta in un quadrato di 256 m di lato, ma è stato spostato il nodo master. In termini matematici l'insieme dei punti che forma la rete si dice *stellato* e, nel primo caso, il nodo master è un possibile centro dello stellato, mentre nel secondo caso, non lo è. Per entrambe le reti i percorsi trovati sono buoni, benché variando la posizione del master si è determinato, nel secondo caso, un piccolo aumento dei tempi di esecuzione. Questo perché, probabilmente, soprattutto nelle prime fasi di evoluzione, il programma è dovuto ricorrere più volte all'algoritmo di Dijkstra per la creazione delle giunzioni.

In tutti questi esempi si è preso in considerazione sempre il percorso di broadcast, vediamo ora come si comporta l'algoritmo nel caso di percorsi di multicast, ovvero quando solo alcuni dei nodi della rete devono essere informati.

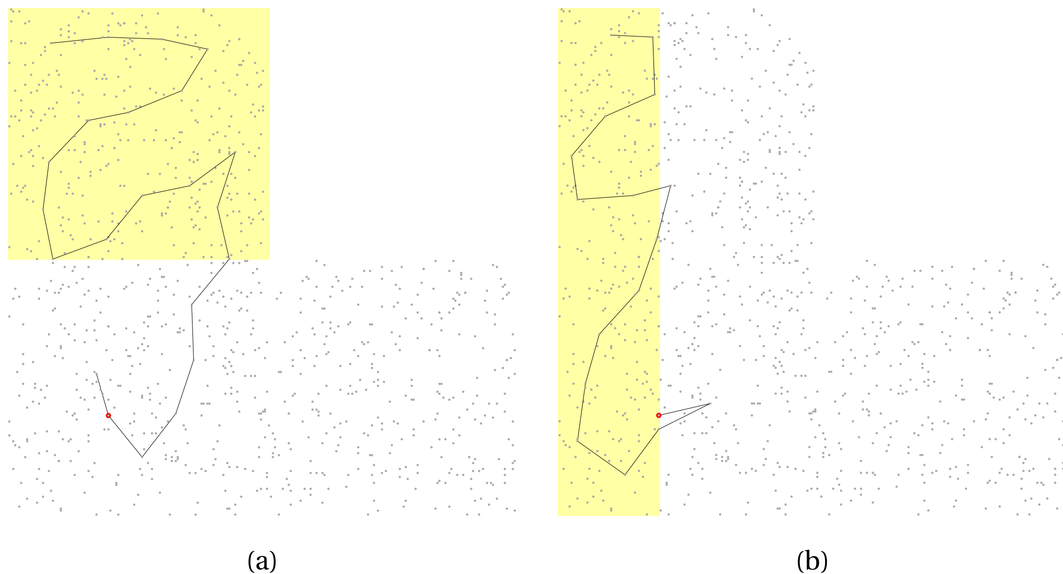


Figura 4.4: *Domini di multicast di prova*

La rete è la stessa di fig. 4.3 e i domini di multicast sono composti dai nodi all'interno delle aree evidenziate. Come si può vedere, in entrambi i casi, l'algoritmo ha generato dei percorsi buoni. I tratti di percorso fuori dal dominio da informare sono brevi, mentre all'interno del dominio viene esplorata tutta la superficie.

Tutti i percorsi mostrati sono stati trovati in tempi relativamente brevi, mediamente inferiori al minuto, utilizzando una macchina con un processore INTEL CORE I7-6500U. Poiché il calcolo di ogni percorso opera su set di informazioni indipendenti è possibile parallelizzare il calcolo riducendo di molto i tempi di esecuzione.

4.2 Simulazione del packet loss

Per stimare la perdita dei pacchetti durante l'attraversamento della rete è stato calcolato il PER (eq. (3.3)) in corrispondenza di ogni salto del percorso, per poi calcolare la probabilità di errore cumulativa su tutto il percorso.

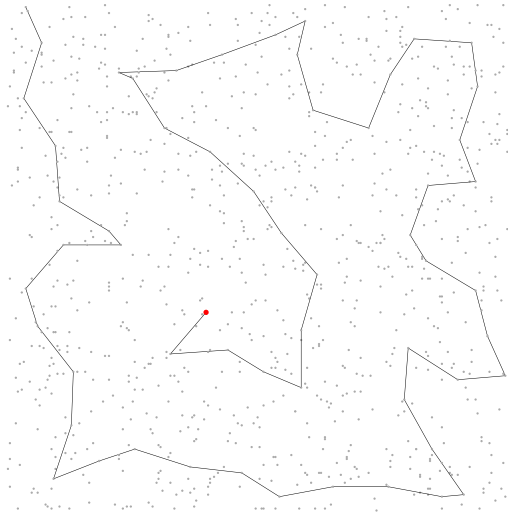
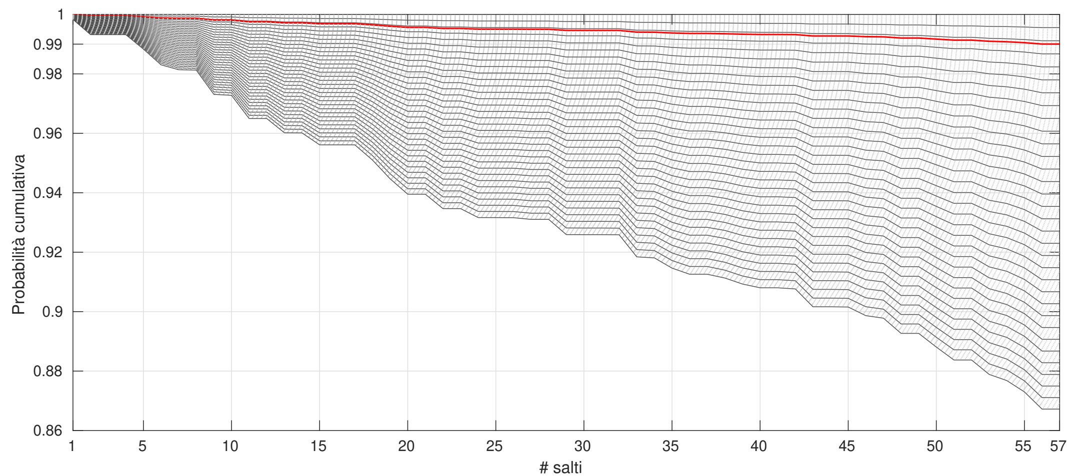


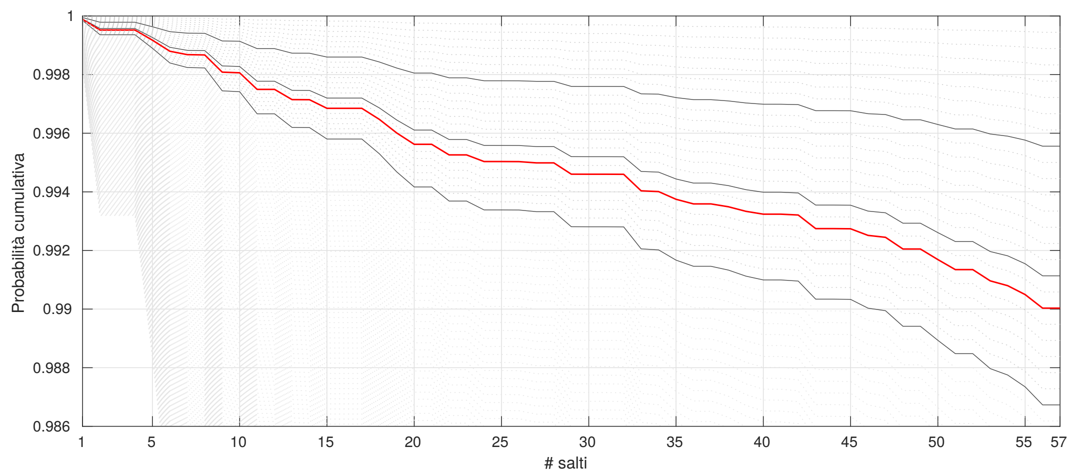
Figura 4.5: *Percorso analizzato*

Il percorso analizzato riportato in fig. 4.5 è quello trovato dall'algoritmo nel caso di rete quadrata con 1000 nodi (fig. 4.1a), è formato da 57 salti ed il salto più lungo è di 29,7 m, quindi ogni salto ha una probabilità di errore sul pacchetto minore, anche di molto, del valore considerato accettabile di 1 %, ma ovviamente la probabilità di errore si accumula ad ogni ritrasmissione. L'informazione potrebbe quindi andare persa mentre il pacchetto si propaga nella rete.

Le curve tracciate di seguito sono state ottenute valutando la probabilità di errore cumulativa per tutte le possibili lunghezze, in byte, del percorso. La curva con probabilità di errore minore (quella più in alto) corrisponde a pacchetti di lunghezza pari a 1 byte. Andando verso il basso aumenta la lunghezza del pacchetto: la curva con l'errore massimo corrisponde a pacchetti di 256 byte. Le curve tracciate in nero sono per le lunghezze multiple di 8 byte, mentre in rosso viene evidenziata quella che corrisponde al pacchetto più lungo con probabilità di errore minore della soglia imposta di 1 %.



(a)



(b)

Figura 4.6: *Probabilità di ricezione corretta cumulativa per ogni salto. a) Pacchetti da 1 a 256 byte di lunghezza; b) pacchetti da 1 a 24 byte di lunghezza, ingrandimento*

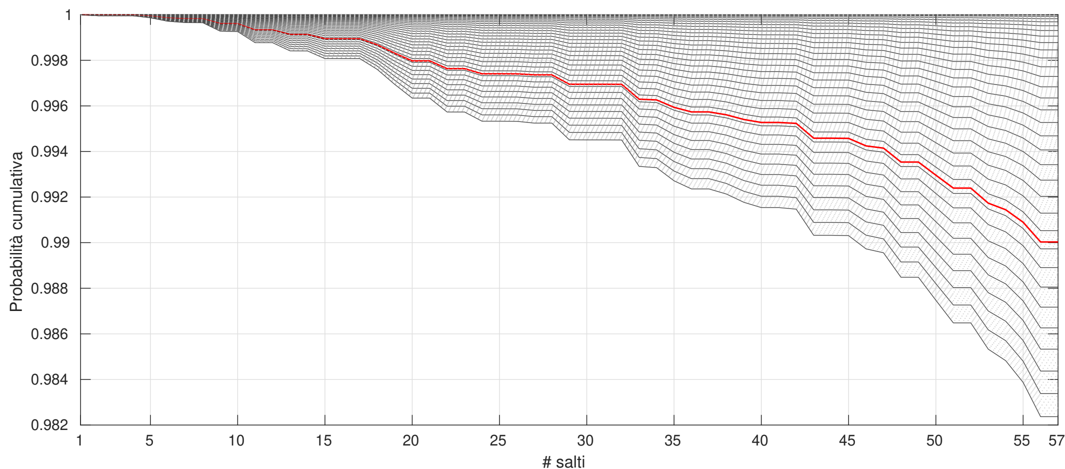
Come si può vedere in fig. 4.6b, per questo particolare percorso, solo i pacchetti non più lunghi di 18 byte (curva evidenziata in rosso) presentano una probabilità d'errore sul singolo bit minore di 1 %. Per i pacchetti di lunghezza massima (256 byte) questa probabilità aumenta fino a circa il 13 %. Questa situazione puù essere migliorata qualora venissero implementate delle strategie di controllo e correzione errori ai pacchetti. Poiché la probabilità che avviene un errore nella ricezione del pacchetto è indipendente dalla presenza di altri errori, si può

dire che la probabilità che ci siano n errori sia pari a:

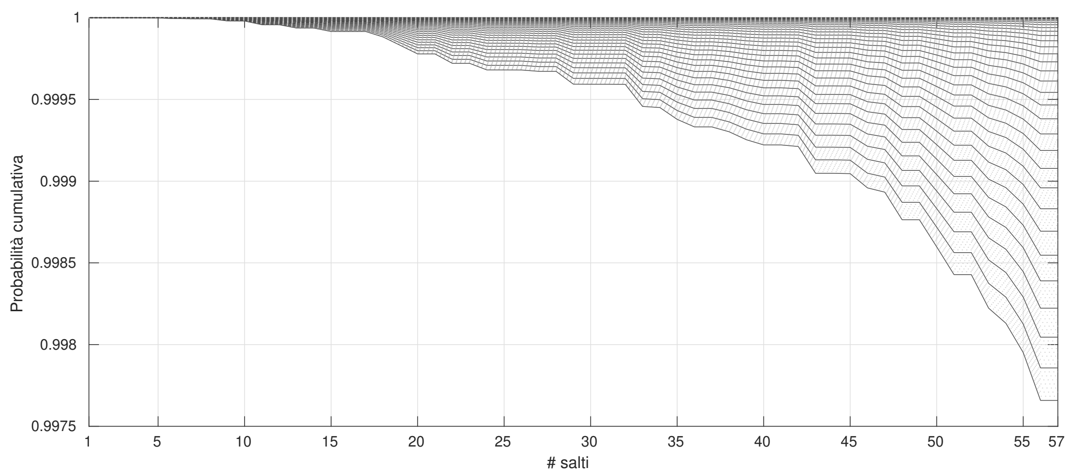
$$P_{\text{n errori}} = PER^n \tag{4.1}$$

Quindi, se si ha la possibilità di individuare e correggere $n-1$ bit per pacchetto, la probabilità di ricezione corretta sarebbe $1 - PER^n$.

Come si può vedere dai grafici in fig. 4.7 la probabilità di ricezione con al massimo un errore ($n = 2$) è garantita con probabilità del 99 % per i pacchetti di lunghezza fino a 189 byte, mentre considerando al massimo due errori ($n = 3$) la probabilità di ricezione corretta pari al 99 % è garantita per tutti i pacchetti.



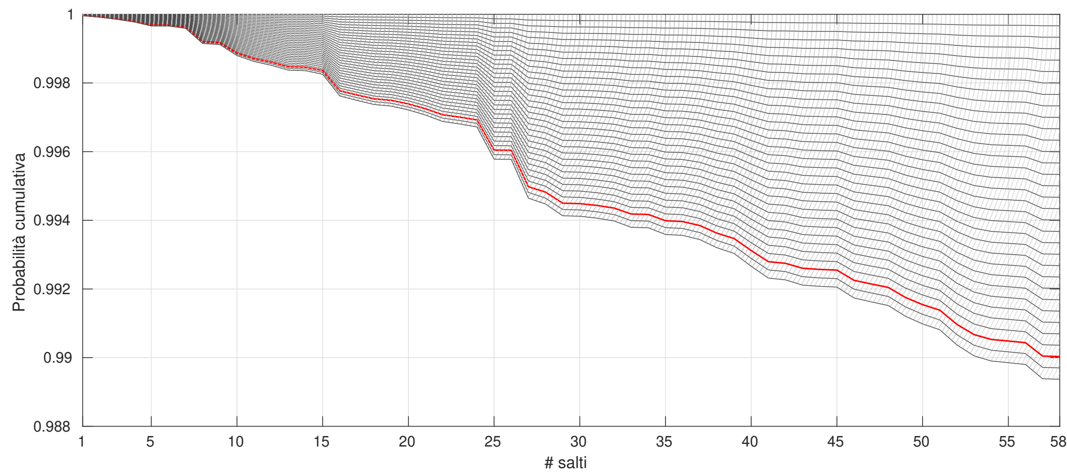
(a)



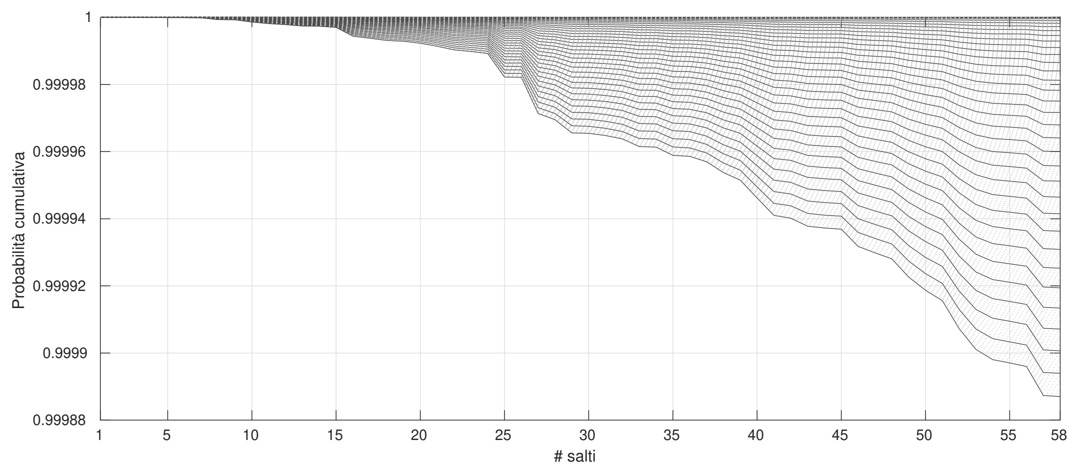
(b)

Figura 4.7: *Probabilità cumulativa per ogni salto di ricezione con al più a) un errore, b) due errori*

Un'altra possibile soluzione per ridurre la probabilità di perdita del pacchetto lungo la rete è di ridurre l'ampiezza dei salti che formano il percorso. Il percorso di fig. 4.5 preso in esame ha una lunghezza complessiva di 1430 m, se si suppone di poter costruire un nuovo percorso che abbia all'incirca la stessa lunghezza e lo stesso tracciato ma composto da salti mediamente più corti, è possibile che la riduzione della probabilità d'errore dovuta ai salti più corti sia molto più influente dell'aumento dovuto al maggior numero di ritrasmissioni.



(a)



(b)

Figura 4.8: *Probabilità cumulativa per un percorso composto da salti di ampiezza media di* $25m$ *nel caso di a) nessun errore correggibile, b) un errore correggibile*

Per coprire una distanza di 1430 m con incrementi di 25 m sono necessari 58 salti, quindi solo uno in più rispetto al percorso generato dall'algoritmo. In fig. 4.8 si possono vedere le prestazioni di un percorso composto da 58 salti di ampiezza distribuita normalmente con media pari a 25 m e varianza unitaria. Per questa particolare estrazione casuale, senza correzione di errori, la probabilità di errore è minore dell'1 % per tutti i pacchetti di lunghezza inferiore a 240 byte. Implementando invece un codice di correzione errori con una capacità di correzione pari ad un bit, la probabilità di errore per il pacchetto di lunghezza massima scende a 120 ppm.

Ovviamente tutti i valori e le quantità riportate in questi esempi sono riferiti allo specifico caso. Comunque si è visto, da altre prove analoghe non riportate per brevità, che sono risultati attendibili e consistenti.

5 Conclusioni

Gli obiettivi principali di questo lavoro di tesi consistevano nell'estensione del protocollo ToLHnet al supporto di comunicazioni di tipo broadcast e multicast, nello sviluppo del firmware per dimostrare le capacità delle nuove strategie di instradamento e nello sviluppo di un algoritmo per la generazione di percorsi di broadcast e multicast.

Per quanto riguarda i primi due punti le strategie implementate appaiono robuste e, dall'implementazione del firmware dei nodi, non sono emerse criticità. I nodi rispondono come previsto, i nuovi pacchetti di configurazione vengono correttamente processati dai nodi, i pacchetti multicast vengono inoltrati e accettati correttamente e i pacchetti vengono filtrati anche in base al gruppo. Pertanto la nuova revisione del protocollo può essere considerata pronta per una fase di beta testing.

Per la generazione dei percorsi si è deciso di implementare un algoritmo genetico. Fin dalle fasi preliminari dello sviluppo si è deciso di non discostarsi dalla definizione più semplice di algoritmo genetico, pertanto ogni decisione presa dall'algoritmo sviluppato è subordinata ad estrazioni casuali, quindi il comportamento dell'algoritmo rientra nella definizione di *computazione evolutiva*. Ci si può ritenere soddisfatti dei risultati ottenuti. In tutti i casi testati l'algoritmo ha generato sempre ottimi percorsi, però sono state prese in considerazioni sempre reti composte da un singolo mezzo trasmissivo. Le prestazioni potrebbero non essere mantenute nel caso di reti eterogenee e, in tal caso, potrebbe rendersi necessaria la riscrittura di tutte le funzioni dove vengono generati segmenti di percorso, in particolare si ritiene che possano presentare maggiormente problemi le funzioni delle mutazioni e la funzione `generateJunction`. Un ultima nota è sui tempi di esecuzione: come detto, eseguendo l'algoritmo su un pc non particolarmente potente i tempi sono dell'ordine di alcune decine di secondi per la generazione di ogni percorso, ma il calcolo dovrà essere eseguito dal nodo master della rete che, benché molto più potente dei nodi periferici, è sensibilmente più limitato in confronto ad un computer. Realisticamente il nodo master potrebbe essere un single board computer (per esempio un *Raspberry Pi* o un *Asus Tinker Board*), in questo caso i tempi di esecuzione potrebbero essere superiori di cinque o dieci volte rispetto ad un computer di media potenza. Si consiglia quindi di implementare il calcolo parallelo dei vari percorsi.

Appendice

A Generatore pseudorandom con PDF non uniforme

In sezione 3 è stata utilizzata più di una volta la funzione `rand()`, che restituisce un numero pseudo-casuale con distribuzione uniforme, per generare una variabile aleatoria discreta con distribuzione non uniforme.

Si supponga di aver un processo aleatorio con quattro possibili esiti con probabilità:

$$\left\{A = \frac{1}{2}; \qquad B = \frac{1}{4}; \qquad C = \frac{1}{8}; \qquad D = \frac{1}{8}\right\}$$

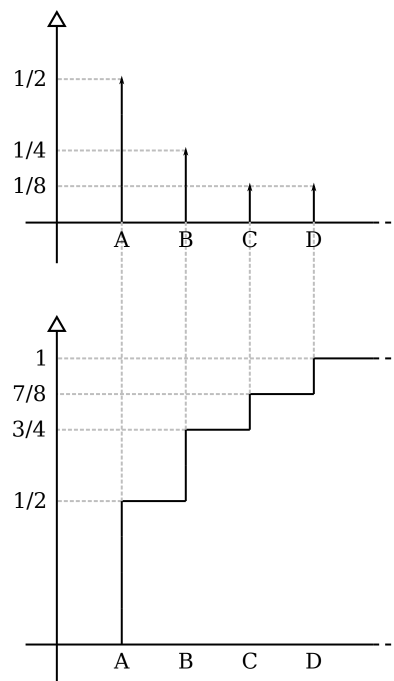


Figura A.1: *Funzione di probabilità e funzione di probabilità cumulativa di una variabile discreta*

Ordinando i quattro possibili esiti dell'esperimento aleatorio su di un asse, la funzione di probabilità cumulativa assume, in ogni punto, il valore della somma della porzione di funzione di probabilità che questo lascia alla sua sinistra.

Quindi, noto il vettore delle probabilità associate ad ogni evento, è possibile calcolare il vettore delle probabilità cumulative:

$$\begin{bmatrix} \frac{1}{2} & \frac{1}{4} & \frac{1}{8} & \frac{1}{8} \end{bmatrix} \Rightarrow \begin{bmatrix} 0 & \frac{1}{2} & \frac{3}{4} & \frac{7}{8} & 1 \end{bmatrix}$$

Estraendo un valore tra 0 e 1, tramite un generatore pseudo-random con distribuzione uniforme, si può associare a questo valore uno degli eventi con probabilità non uniforme confrontando il valore estratto con il vettore delle probabilità cumulative: l'evento è associato all'intervallo tra due valori contigui del vettore. In altre parole quello che si sta facendo è valutare l'inverso della funzione di probabilità cumulativa in corrispondenza di punti casuali estratti uniformemente nell'intervallo $[0;1]$.

B Struttura dei dati utilizzati nell'algoritmo genetico

```
class point
{
public:
  double x, y;
  point(double x = 0, double y = 0);
  std::pair<double, double> getCart();
  std::pair<double, double> getPolar(double x0 = 0, double y0 = 0);
  std::pair<double, double> getPolar(point p0);
  double dist(double x0 = 0, double y0 = 0);
  double dist(point p0);

  point operator+(point &);
  point operator-(point &);

private:
};
```

La classe *point*, come già detto, contiene , negli attributi x e y, le coordinate cartesiane di un punto sul piano. Il metodo *getCart()* è il getter che restituisce i valori di x e y. Il metodo *getPolar(point p0)* restituisce le coordinate polari dell'oggetto riferite al polo *p0*. Il metodo *dist(point p0)* restituisce la distanza dell'oggetto dal punto *p0*. Sono stati, inoltre, ridefiniti gli operatori $+$ e $-$ per implementare la somma vettoriale in $\mathbb{R}^2$ e mantenere più pulita l'implementazione di *getPolar* e *dist*.

```
typedef std::vector<vertex_descriptor> individual_path;

class individual
{

public:
  graph_t *g;
  std::vector<point> *pos;
  individual_path path;
  std::vector<bool> to_inf;
  std::vector<bool> informed;
  unsigned informed_qty;
  double fitness;

  std::vector<double> bonuses;
  std::vector<double> maluses;

  enum origin{
    initial,
    prev,
    cross1,
    cross2
  } source;

  individual(individual_path a, graph_t *g, std::vector<point> *poss,
   ↪  std::vector<bool> to_inf);
  void calcFitness();

  void print();
  void dump(const char *basename);

  bool operator==(const individual &t);

private:
  unsigned unique_nodes;
};
```

La classe `individual` rappresenta un individuo dell'algoritmo genetico. In `path` è salvato il vettore di nodi che formano il percorso. Il vettore `to_inf` è la bitmap che rappresenta i nodi che devono essere informati dal percorso, mentre `informed` è la bitmap che rappresenta i nodi effettivamente informati dal percorso, la quantità di questi nodi è salvata anche in `informed_qty`. In `fitness` è salvato il valore di fitness relativo all'individuo e nei vettori `bonuses` e `maluses` sono salvati i valori intermedi utili nel corso dell'algoritmo. Il valore di `source` indica l'origine dell'individuo. Il metodo `print()` serve a scrivere nello `stdout` un riassunto dell'individuo, mentre `dump(char)` serve a salvare su file le coordinate dei nodi del percorso e il vettore `bonuses`.

```
class broadcast_handler
{
public:
  std::string name;
  std::vector<individual> offspring;
  std::vector<individual> population;
  std::vector<bool> to_inf;

  broadcast_handler(std::string name, graph_t &g,
   ↪  std::deque<node_description> &nodes, vertex_descriptor &root,
   ↪  std::vector<vertex_descriptor> &parents, std::vector<bool>
   ↪  to_inf);

private:
  graph_t graph;
  unsigned initial_pop;
  unsigned max_population;
  unsigned max_offspring;
  double mutation_chance;
  std::deque<node_description> *nodes_m;
  std::vector<point> poss;

  std::vector<unsigned> initialPopulationEndpoints(vertex_descriptor
   ↪  &root);
  void generateInitialPopulation(vertex_descriptor &root,
   ↪  std::vector<vertex_descriptor> &parents);
  void crossover(individual &a, individual &b);
  void crossover2(individual &a, individual &b);
  void generation();
  std::vector<vertex_descriptor> createJunction(vertex_descriptor,
   ↪  vertex_descriptor);
  vertex_descriptor findNextNode(vertex_descriptor, vertex_descriptor);
  individual mutation_ret(individual&);
  individual mutation1_ret(individual&);

  bool parse_conf(const char * filename);

#ifdef DUMP_ROUTING_GRAPH
  void plot_pops(const char *filename, bool trace_unused = false,
   ↪  double scale = 1);
  void plot_individual(const char *filename, individual_path &a, bool
   ↪  trace_unused = false, double scale = 1);
  void plot_individual(const char *filename, individual_path &a,
   ↪  graph_t &g, bool trace_unused = false, double scale = 1);
  void plot_individual(const char *filename, individual_path &a,
   ↪  graph_t &g, vertex_descriptor, vertex_descriptor, double scale =
   ↪  1);
#endif
};
```

Infine, la classe *broadcast_handler* è la classe principale dell'algoritmo genetico, in essa sono definite tutte le funzioni che costituiscono l'algoritmo. Poiché durante l'esecuzione del programma ci sarà un oggetto di questa classe per ogni dominio di multicast, gli attributi pubblici sono utili per identificare ogni

oggetto. L'attributo `initial_pop` contiene la quantitò di individui della popolazione iniziale, `max_population` la quantità massima dell'insieme della popolazione, `max_offspring` la quantità massima degli individui discendenti di una popolazione, `mutation_chance` la probabilità di mutazione. Questi parametri determinano il comportamento dell'algoritmo evolutivo e vengono inizializzati dal metodo `parse_conf(char)`. I metodi `plot_pops` e `plot_individual` servono a salvare l'immagine dei percorsi in formato DOT.

Riferimenti bibliografici

[1] G. Biagetti, P. Crippa, A. Curzi, S. Orcioni e C. Turchetti, «ToLHnet: A low-complexity protocol for mixed wired and wireless low-rate control networks», in *2014 6th European Embedded Design in Education and Research Conference (EDERC)*, set. 2014, pp. 177–181. DOI: 10.1109/EDERC.2014.6924383.

[2] J. N. Al-Karaki e A. E. Kamal, «Routing techniques in wireless sensor networks: a survey», *IEEE Wireless Communications*, vol. 11, n. 6, pp. 6–28, dic. 2004. DOI: 10.1109/MWC.2004.1368893.

[3] K. Akkaya e M. Younis, «A Survey on Routing Protocols for Wireless Sensor Networks», *Ad Hoc Networks*, vol. 3, pp. 325–349, mag. 2005. DOI: 10.1016/j.adhoc.2003.09.010.

[4] A. Vieira. (2018). GNU Arm Embedded Toolchain PPA, indirizzo: https://launchpad.net/~team-gcc-arm-embedded/+archive/ubuntu/ppa (visitato il 13/09/2019).

[5] Texas Instruments. (2017). TivaWare for C Series, indirizzo: http://www.ti.com/tool/SW-TM4C (visitato il 13/09/2019).

[6] G. Biagetti. (2014). ToLHnet technical specification, indirizzo: http://www.tolhnet.org/download.php?f=tolhnet-spec.pdf (visitato il 13/09/2019).

[7] M. Kumar, M. Husain, N. Upreti e D. Gupta, «Genetic algorithm: Review and application», *Journal of Information & Knowledge Management*, lug. 2010.

[8] J. H. Holland, *Adaptation in Natural and Artificial Systems*. University of Michigan Press, 1975.

[9] C. Lomont. (2003). Fast inverse square root, indirizzo: https://www.lomont.org/papers/2003/InvSqrt.pdf (visitato il 05/10/2019).

[10] D. Ausili, «Implementazione di strategie di indirizzamento multicast nel protocollo ToLHnet mediante algoritmi genetici», tesi di laurea mag., Università Politecnica delle Marche, 2015.