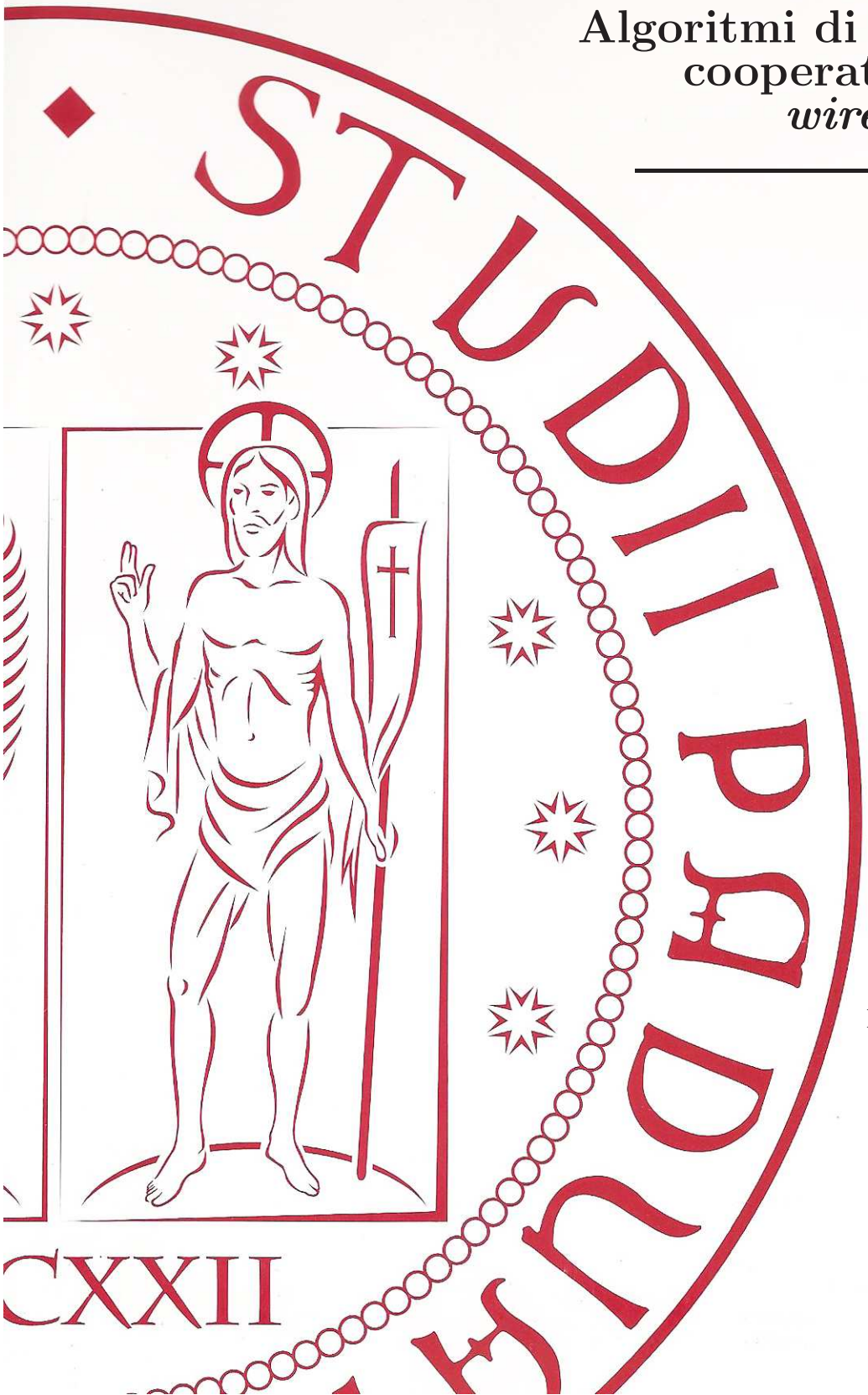
Università degli Studi di Padova

DEI Dipartimento di Ingegneria dell'Informazione

Corso di Laurea in Ingegneria dell'Informazione

Tesina

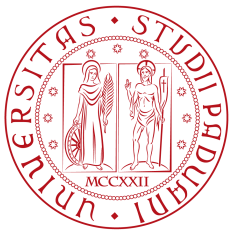
Algoritmi di *pathfinding* cooperativo per reti *wireless ad hoc*



Laureando:
Niccolò Franceschi

Relatore:
Prof. Stefano Tomasin

23 luglio 2010
A.A. 2009/2010



Corso di Laurea in Ingegneria dell'Informazione

Tesina

Algoritmi di *pathfinding* cooperativo per reti *wireless ad hoc*

Laureando:
Niccoló Franceschi

Relatore:
Prof. Stefano Tomasin

23 luglio 2010
A.A. 2009/2010

Indice

Sommario

Negli ultimi anni le problematiche di progettazione ed implementazione di reti *wireless ad hoc* hanno acquistato crescente attenzione nel campo delle telecomunicazioni. Una rete *ad hoc* è una rete *wireless* decentralizzata che non si appoggia su strutture preesistenti come *router* o *access point*, ma è gestita con la partecipazione dei dispositivi di cui è composta. Per sua natura una rete *ad hoc* deve garantire la cooperazione e la collaborazione dei nodi ed è progettata con approccio multiutente. In particolare i vari dispositivi saranno tenuti ad unire le proprie forze per instradare i pacchetti di dati che devono essere scambiati da un nodo all'altro della rete.

In questa tesina ci concentreremo sulle problematiche che riguardano il *pathfinding* cooperativo, cioè quella fase preliminare o parallela alla trasmissione in cui si cercano dei percorsi ammissibili per tutti i pacchetti che simultaneamente viaggiano nella rete *ad hoc*. Per prima cosa si cercherà un modello per la rete e il problema verrà ricondotto alla teoria dei grafi, successivamente sarà introdotto l'algoritmo A*, utile per cercare cammmini di costo minimo. In seguito analizzeremo alcuni lavori nati nell'ambito dell'Intelligenza Artificiale e si cercherà di capire in che modo si possano adattare a problematiche di *pathfinding* nelle reti *ad hoc*. Particolare rilievo sarà riservato all'*overhead* di ogni soluzione proposta e alle complicazioni provocate dall'interferenza dei dispositivi. I metodi studiati vengono raggruppati in base alla quantità di informazione condivisa tra i dispositivi: distingueremo tra approcci *centralizzati* in cui il *pathfinding* viene gestito globalmente con un'azione unitaria di tutti i nodi, e soluzioni *distribuite* che si prefiggono di raggiungere la cooperazione tra i nodi senza appesantire ulteriormente l'*overhead*.

Capitolo 1

Un modello per il sistema

Per prima cosa cerchiamo un modello adatto per descrivere le reti *ad hoc* e riconduciamo il *pathfinding* ad un problema risolubile con la Teoria dei grafi. Consideriamo una rete *wireless* formata da N nodi, distribuiti nello spazio in modo qualsiasi. Ogni nodo è capace di relazionarsi con l'esterno attraverso una antenna radio. Immaginiamo che la variabile temporale sia quantizzata in tanti slot, che rappresentano l'unità minima in cui un pacchetto dati è trasmesso. L'inizializzazione della rete suddivide i nodi in *cluster*, gruppi di dispositivi in grado di cooperare tra loro. La collaborazione tra più nodi avviene attraverso una trasmissione simultanea dello stesso pacchetto di dati nel medesimo slot di tempo, operazione che alza il limite di Shannon per quella specifica comunicazione. Ecco quindi che la cooperazione tra vari nodi permette, poiché coinvolge più antenne, di trasmettere ad un bit rate più alto senza perdere il pacchetto.

1.1 Grafo dei dispositivi

Modelliamo la nostra rete ad un maggiore livello di astrazione: costruiamo un grafo orientato $G = (V, E)$ in cui V sia l'insieme dei vertici ed E quello degli archi. Ogni vertice del grafo è un *nodo virtuale*, che non necessariamente coincide con un nodo della rete reale. Infatti i nodi virtuali possono essere di tre tipi:

T1 un singolo nodo della rete,

T2 un cluster di nodi,

T3 un sottoinsieme di nodi appartenenti ad un cluster

Gli archi $(i, j) \in E$ rappresentano una possibile trasmissione tra i nodi virtuali i e j in un determinato time slot.

Ad ogni arco (i, j) associamo un costo c_{ij}, che tiene conto dell'energia spesa nella trasmissione, del suo ritardo e dell'affidabilità del collegamento:

$$c_{ij} = \begin{cases} c & \text{se } i = j \\ \frac{\beta c + (1-\beta) w_i}{p_i j} & \text{se } i \neq j \end{cases} \tag{1.1}$$

con: c ritardo nella trasmissione, w_i numero di nodi realmente presenti nel nodo virtuale i, p_{ij} probabilità di corretta ricezione di un pacchetto, parametro $\beta \in [0, 1]$. Per scoraggiare soluzioni ottime del problema che contengano archi degeneri non strettamente necessari, si è inserito il caso $i = j$, situazione in cui il nodo non comunica con gli altri durante lo slot di tempo.

Ai fini di ottenere una formulazione risolvibile con algoritmi di Programmazione Lineare (PL), definiamo *richiesta* una coppia di nodi (s, d), con $s, d \in V$ e $s \neq d$, che rappresentano i nodi di sorgente e destinazione nell'instradamento di un singolo pacchetto; l'insieme di tutte le richieste possibili è

$$\mathcal{D} = \{(s_1, d_1), (s_2, d_2), \ldots, (s_K, d_K)\} \tag{1.2}$$

Diremo che un sottografo $H = (V', E')$ di G connette una richiesta (s, d), se è possibile individuare un cammino $\mathcal{P}$ in H che colleghi s a d, cioè esista

$$\mathcal{P} = \{(s, n_1), (n_1, n_2), \ldots, (n_l, d)\} \subseteq E' \tag{1.3}$$

Ciascun arco in $\mathcal{P}$ è una comunicazione tra due nodi in un diverso slot temporale, cosicché, se il cammino consta di $l + 1$ archi, saranno necessari almeno $l + 1$ slot per spedire un pacchetto tra s e d. Si noti che alcuni degli archi potrebbero essere degeneri, cioè il pacchetto poterebbe sostare per uno o più slot di tempo nello stesso nodo col fine di evitare di interferire con altre trasmissioni.

1.2 Grafo di interferenza

Vogliamo che il modello che stiamo costruendo tenga conto dell'interferenza tra i vari nodi in comunicazione nello stesso time slot, per farlo immaginiamo che durante una trasmissione ogni nodo coinvolto si trovi al centro di una *sfera di interferenza* entro cui non è permessa alcuna ulteriore comunicazione. Dati due nodi $i, j \in V$, indichiamo con d_{max} la massima distanza entro la quale un pacchetto spedito da i arriva a j con probabilità maggiore o uguale a ϵ, possiamo allora definire il raggio di interferenza di un qualsiasi nodo come αd_{max}, con $\alpha \geqslant 1$ parametro della rete.

Diamo un criterio formale di interferenza tra due trasmissioni: siano $n_i, n_j, n_h, n_k \in V$ quattro nodi del grafo, indifferentemente ciascuno potrà essere dei tre tipi T1, T2, T3 visti

in precedenza, diciamo allora che la comunicazione $n_i \to n_j$ interferisce con $n_h \to n_k$ se almeno una delle due condizioni è verificata:

C1 Esiste una coppia di dispositivi reali (a, b), con a in n_i e b in n_k tale che $d(a, b) < \alpha d_{max}$, in questo caso la trasmissione da n_i interferisce con la ricezione in n_k.

C2 Esiste una coppia di dispositivi reali (a, b), con a in n_h e b in n_j tale che $d(a, b) < \alpha d_{max}$, in questo caso la trasmissione da n_h interferisce con la ricezione in n_j.

A questo punto abbiamo gli strumenti per creare un modello per l'interferenza della rete che si presti ad essere risolto con algoritmi di programmazione lineare. Definiamo un *grafo di interferenza* $I = (V, E_i)$ che abbia gli stessi vertici di G, mentre gli archi (n_i, n_j) sono ottenuti collegando n_i a n_j solo se esistono due nodi reali, il primo in n_i ed il secondo in n_j, che distano meno di αd_{max}. Per una rappresentazione dei due grafi appena introdotti, si veda la Figura 1.1.

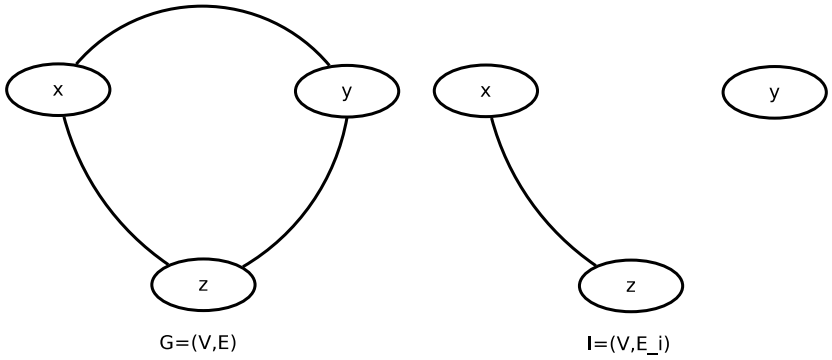


Figura 1.1: I grafici dei dispositivi e di interferenza

Capitolo 2

L'algoritmo A^*

Prima di studiare delle soluzioni cooperative per il problema del *pathfinding* in una rete di dispositivi, dobbiamo dotarci di strumenti teorici che ci permettano di cercare un cammino di costo minimo all'interno del grafo con cui abbiamo modellato la rete. Poiché la presenza di più richieste all'interno della stessa rete complica notevolmente la trattazione, inizieremo il nostro studio occupandoci del caso più semplice, in cui c'è una sola richiesta da soddisfare. Sappiamo che, anche in queste ipotesi, trovare un cammino di costo minimo all'interno di un grafo orientato è un problema NP-difficile, tuttavia, se assumiamo che nessuno degli archi abbia peso negativo, è possibile risolvere questo problema in tempo polinomiale. La complessità computazionale di una soluzione, unitamente al suo grado di ottimalità, costituirà un elemento di valutazione per gli algoritmi che incontreremo nei prossimi paragrafi. Introduciamo ora A*, un algoritmo ben noto che sarà usato come base per affrontare il *pathfinding* cooperativo.

A^* è un algoritmo che risolve in modo ottimo il problema della ricerca di cammini minimi. La sua velocità è maggiore rispetto all'algoritmo di Dijkstra (vedi [1]) nel caso in cui non sia importante calcolare il percorso da una sorgente a tutti gli altri nodi del grafo, ma interessi trovare la strada verso un singolo nodo destinazione. Rispetto a Dijkstra, infatti, A^* non visita tutti i nodi, ma *prevede* quelli che lo porteranno più vicino alla meta. L'algoritmo opera questa previsione attraverso un'*euristica*, cioè una stima del costo della distanza rimanente tra il nodo in fase di elaborazione e quello finale. Una volta che conosce quali sono i nodi più vicini alla destinazione, A^* orienta le sue ricerche e procede fino alla fine senza perdere tempo ad elaborare nodi non necessari. Ribadiamo che A^* cerca il cammino minimo per singole coppie sorgente-destinazione, quindi non è immediatamente applicabile al nostro problema, in cui dobbiamo gestire $|\mathcal{D}|$ richieste contemporaneamente.

E' chiaro che l'efficacia dell'algoritmo dipende essenzialmente dalla bontà dell'euristica scelta. In ogni caso l'euristica deve essere una stima per difetto della distanza, pena la possibilità che il cammino trovato non sia minimo, ma non potrà rappresentare un vincolo troppo lasco rispetto al costo reale, altrimenti l'algoritmo avrebbe prestazioni insoddisfa-

centi. Diciamo che un'euristica è *ammissibile* quando stima una distanza sempre e solo per difetto. Dobbiamo quindi cercare una compromesso tra ammissibilità e accuratezza dell'euristica: il suo valore deve tendere a quello reale senza mai superarlo. Il caso ideale è quello in cui l'euristica fornisce l'esatto valore del costo del cammino rimanente, in tal caso l'algoritmo giunge a destinazione nel minor numero di passi possibile, come se sapesse a priori la strada verso la fine. Il caso peggiore è quello in cui la distanza stimata è sempre nulla, in tal caso A* diventa l'algoritmo di Dijkstra. Vedremo poi due euristiche molto semplici.

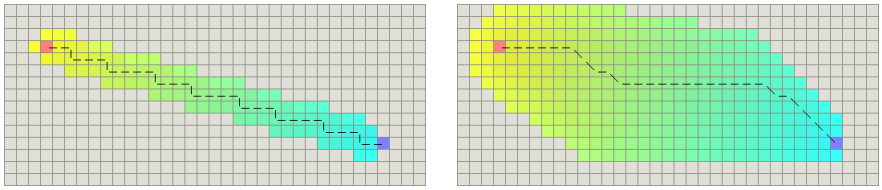


Figura 2.1: Due euristiche a confronto. I nodi colorati sono quelli inseriti nella lista Open, quelli grigi non vengono mai elaborati.

Una possibile realizzazione di A^* in pseudo-codice è riportata nel Codice 2.1. Entrando nei dettagli implementativi, si fa uso di due liste *Open* e *Closed*, rispettivamente per i nodi da elaborare e per quelli di cui già si conosce il cammino minimo, e si associano ad ogni nodo i quattro campi, $pred$[i], g[i], h[i], f[i]:

pred per i nodi nella lista Closed è il predecessore di i nella sequenza di nodi che forma il cammino minimo, per i nodi nella lista Open, invece, è il nodo che fino a quel momento risulta essere il predecessore, ma potrebbe cessare di esserlo in seguito a successive elaborazioni.

g per i nodi nella lista Closed è il costo del cammino minimo dal nodo di partenza a i, mentre per quelli nella lista Open è quello che fino a quel momento risulta essere il costo minimo

h è una stima del costo del cammino da i al nodo destinazione calcolata con un'euristica,

f è la somma di g ed h: $f[i] = g[i] + h[i]$.

L'algoritmo di volta in volta sceglie il nodo da inserire nella lista Closed sulla base del valore della funzione g, poi procede ad aggiornare i valori di *pred, g, f* ed h per tutti i nodi adiacenti. L'esecuzione ha termine con successo quando il nodo destinazione è inserito in Open, mentre la ricerca fallisce se la lista Open si svuota (tutti i nodi sono stati elaborati e nessuno di questi era quello di arrivo). La procedura CAMMINO(nodo) è ausiliaria al corpo principale di A* e serve a ricostruire a ritroso il cammino minimo una volta che è stato trovato.

```
Algoritmo A*(s,d)
  Closed ⟵ ∅
  Open ⟵ s
  g[s] ⟵ 0
  h[s] ⟵ EURISTICA(s,d)
  f[s] ⟵ h[s]
  pred[s] ⟵ s
  while Open ≠ ∅
      k ⟵ argmin{f[i], i ∈ Open}
      if k = d
          return CAMMINO(pred[d])      /*Successo*/
      Open ⟵ Open ∖ {k}
      Closed ⟵ Closed ∪ {k}
      for each j ∉ Closed | (k,j) ∈ E /*nodi limitrofi*/
          if j ∉ Open
            Open ⟵ Open ∪ {j}
            g[j] ⟵ g[k] + c_kj
            h[j] ⟵ EURISTICA(j,d)
            f[j] ⟵ g[j] + h[j]
            pred[j] ⟵ k
          else if (g[j] > g[k] + c_kj)
            g[j] ⟵ g[k] + c_kj
            f[j] ⟵ g[j] + h[j]
            pred[j] ⟵ k
  return insuccesso /*Insuccesso*/

Algoritmo CAMMINO(nodo)
  if pred[nodo] = nodo
       return nodo
  else return {nodo} ∪ CAMMINO(pred[nodo])
```

Codice 2.1: Elementare implementazione di A^* in pseudo-codice. Cerca un cammino di costo minimo tra i nodi s e d del grafo $G = (V, E)$.

Presentiamo ora due regole molto semplici che possono essere usate per implementare la funzione EURISTICA(a,b).

Euristica esatta

Come già anticipato, se ogni coppia di nodi conosce la propria distanza in modo esatto, posso eseguire A* in modo estremamente rapido. L'algoritmo, infatti, inserisce in *Open* un numero esiguo di nodi che elabora rapidamente fino alla meta. Per quanto consentito della struttura e dall'implementazione dei dispositivi che compongono la rete *wireless*, dovremmo sempre cercare di ricondurci a questo tipo di euristica.

$$EURISTICA(j, d) = dist(j, d) \tag{2.1}$$

Nel caso in cui siano presenti più richieste all'interno dello stesso grafo, la distanza esatta diventa una stima per difetto dello spazio realmente percorso, infatti sarà necessario allungare i tragitti per evitare l'interferenza tra cammini.

Euristica Manhattan

Manhattan è un modo molto semplice per costruire un'euristica ammissibile nel caso in cui i nodi non abbiano informazioni dettagliate sulla loro distanza reale. L'idea è quella che ogni nodo conosca le coordinate spaziali di tutti i nodi, e che valuti la distanza semplicemente facendo la differenza delle componenti:

$$EURISTICA(j, d) = |pos_x(j) - pos_x(d)| + |pos_y(j) - pos_y(d)| \tag{2.2}$$

La distanza così ottenuta è un'approssimazione per difetto di quella reale, infatti due dispositivi potrebbero essere topologicamente molto vicini ma non poter comunicare tra di loro a causa di ostacoli o interferenze.

Capitolo 3

Approccio centralizzato

Il problema dell'instradamento cooperativo presenta svariate tecniche di risoluzione, diverse per strumenti usati e per complessità computazionale, tuttavia si possono rintracciare due principali categorie di soluzioni: *centralizzate* e *distribuite*. Le soluzioni centralizzate affrontano la situazione globalmente, cioè tenendo conto di tutte le richieste e cercando un insieme di cammini ottimo che le soddisfi tutte. In [2] vengono presentati due metodi di tipo centralizzato, uno che si avvale dell'algoritmo del *Simplesso*, l'altro di A^* .

3.1 Modello PLI

Come primo approccio scriviamo una formulazione di Programmazione Lineare Intera (PLI) che si potrà risolvere con il metodo del Simplesso [1]. Dato un insieme $\mathcal{D}$ di richieste, il nostro obiettivo è trovare un gruppo di cammini che le connetta tutte e abbia costo minimo. Per ogni richiesta $d \in \mathcal{D}$ chiamiamo i nodi sorgente e destinazione s_d e d_d, mentre per ogni arco $(i,j) \in E$ definiamo la variabile decisionale $x^d_{i,j}(t)$, che vale 1 se il pacchetto associato a d è trasmesso durante lo slot t nel collegamento (i,j), e vale 0 nel caso in cui il collegamento (i,j) non venga coinvolto nello slot t.

La nostra funzione obiettivo (F.O) è:

$$\min \sum_{d \in \mathcal{D}} \sum_{(i,j) \in E} \sum_{t \geqslant 0} c_{(i,j)} x^d_{i,j}(t), \tag{3.1}$$

con questi vincoli:

- I cammini creati sono un sottografo di G connesso, cioè per ogni richiesta c'è una solo nodo che trasmette in ogni time slot t, e tale nodo ha a sua volta ricevuto il pacchetto da un dispositivo raggiungibile nello slot $t-1$:

$$\sum_{(j,h) \in E | x^d_{i,j}(t-1)=1} x^d_{j,h}(t) = 1 \quad \forall d \in \mathcal{D}, \forall t. \tag{3.2}$$

- Se due trasmissioni interferiscono non possono essere attivate contemporaneamente:

$$x_{i,j}^{d}(t)x_{l,m}^{d'}(t) = 0 \quad (l,j),(i,m) \in A \quad d,d \in \mathcal{D} \quad d \neq d' \quad \forall t. \tag{3.3}$$

- I pacchetti devono iniziare il loro percorso dalla sorgente associata alla richiesta:

$$\sum_{(i,j)\in E} x_{i,j}^{d}(t) = 1 \quad d \in \mathcal{D} \quad i = s_d \quad \forall t, \tag{3.4}$$

  e terminare nella destinazione corrispondente:

$$\sum_{(i,j)\in E} x_{i,j}^{d}(t) = 1 \quad d \in \mathcal{D} \quad i = d_d \quad \forall t \tag{3.5}$$

- L'ultimo vincolo della formulazione è quello che dà senso alla variabile decisionale:

$$0 \leq x_{i,j}^{d}(t) \leq 1 \quad \text{interi}, \quad (i,j) \in E \quad d \in \mathcal{D} \quad \forall t. \tag{3.6}$$

Poiché abbiamo una variabile per ogni lato, e il massimo numero di lati in un grafo di n vertici è $\frac{n(n-1)}{2}$, il numero di variabili coinvolte cresce con il quadrato dei nodi e linearmente rispetto al numero di time slot; di conseguenza la risoluzione di questo sistema con un generico software di PL potrebbe essere estremamente onerosa in termini di tempo e di risorse richieste.

3.2 Vettori di stato

Cominciamo ad introdurre il concetto di *stato* del sistema come illustrato in [2], nozione che ci permetterà di trattare il nostro problema in modo indipendente dal numero di richieste e quindi di ricondurci ad una formulazione risolubile con A^* . Chiameremo *vettore di stato* del sistema al generico istante t la K-pla ordinata $\mathbf{a}(t) = (a_1, a_2, \ldots a_K)$, con $a_d \in V$. Ogni componente a_d rappresenta un nodo virtuale a_d che ha ricevuto il pacchetto associato alla richiesta d ed è in grado di trasmetterlo nello slot t. Una transizione di stato da $\mathbf{a}(t) = (a_1, a_2, \ldots a_K)$ a $\mathbf{b}(t) = (b_1, b_2, \ldots b_K)$ è consentita solo se sono soddisfatte entrambe queste condizioni:

1. ognuno dei nodi reali in a_i è raggiungibile da quelli in b_i, cioè $(a_i, b_i) \in E$ con $i = 1, \ldots, K$,
2. non si verifica interferenza, cioè $(a_i, b_i) \notin E_i, \quad \forall i \neq j$

Il costo di una transizione dallo stato **a** allo stato **b** è:

$$c(\mathbf{a} \to \mathbf{b}) = \sum_{i=1}^{k} c_{a_i b_i} \tag{3.7}$$

In questo modo ci siamo ricondotti a cercare il cammino di costo minimo per una singola richiesta dal vettore iniziale $\mathbf{s} = (s_1, \ldots, s_K)$ a quello finale $\mathbf{f} = (f_1, \ldots, f_K)$. Per risolvere questo problema con A^* abbiamo bisogno di definire in modo opportuno un'euristica $h(\mathbf{x})$ che stimi per difetto la distanza che separa $\mathbf{x} = (x_1, \ldots, x_K)$ dallo stato finale $\mathbf{f} = (f_1, \ldots, f_K)$; procediamo come segue:

1. Calcoliamo il cammino minimo tra x_i e f_i attraverso l'algoritmo di Dijkstra, chiamiamo $h(x_i)$ il costo del percorso trovato

2. Ripetiamo per tutte le componenti di $\mathbf{x}$ e otteniamo $h(\mathbf{x})$ da:

$$h(\mathbf{x}) = \sum_{i=1}^{k} h(x_i) \tag{3.8}$$

Gli $h(x_i)$ equivalgono ai cammini minimi che si otterrebbero trascurando l'interferenza, di conseguenza $h(\mathbf{x})$ è un limite inferiore per il costo del cammino reale e l'euristica è *ammissibile*.

La soluzione ottenuta con il metodo dei vettori di stato è certamente ottima a causa dell'ammissibilità dell'euristica, e risolve in modo definitivo anche le problematiche legate all'interferenza: le collisioni tra pacchetti si possono escludere a priori. Certamente questo approccio è il migliore possibile a livello teorico, tuttavia si possono riscontrare alcune problematiche per quanto riguarda l'implementazione:

- Il costo computazionale potrebbe essere estremamente elevato per reti con un numero alto di nodi, ed in generale la dipendenza da N e da $|\mathcal{D}|$ è molto forte.

- Per poter pianificare lo scambio di pacchetti in modo centralizzato devo ipotizzare che almeno uno dei nodi conosca tutte le richieste, si occupi di elaborarle e provveda ad informare tutti i nodi coinvolti nelle trasmissioni.

- Lo stesso nodo che pianifica il *routing* deve conoscere la topologia della rete.

Capitolo 4

Approccio quasi-distribuito

David Silver propone in [3] vari metodi per l'instradamento cooperativo che abbandonano l'approccio *centralizzato* visto fino ad ora, privilegiando invece soluzioni *quasi-distribuite* in cui ogni richiesta è scarsamente influenzata dalle altre. Gli algoritmi contenuti nel suo lavoro sono pensati per coordinare unità in movimento nell'ambito dei videogiochi di strategia, capiremo in che misura possono essere adattati alle telecomunicazioni.

Questi metodi, come si vedrà meglio nella Sezione 4.5, non sono totalmente distribuiti, infatti ogni nodo deve tenere traccia del comportamento di tutti gli altri, cioè deve poter comunicare con loro. In un contesto di reti *ad hoc* questo vincolo può essere gravoso. Nel presentare gli algoritmi faremo ancora riferimento al modello di sistema della Sezione 1.1: sugli archi del grafo $G = (V, E)$ possiamo pensare all'esecuzione di ogni richiesta come al cammino di un *agente* tra il nodo sorgente e quello destinazione. La parola *agente* non nasce nell'ambito delle telecomunicazioni, tuttavia è molto diffusa nella letteratura relativa all'Intelligenza Artificiale e alle problematiche di *pathfinding*, per questo motivo useremo il termine agente ad indicare il soggetto in movimento nel grafo, cioè il pacchetto di informazione che per ogni richiesta deve essere scambiato tra due nodi.

Premettiamo che tutti gli algoritmi che seguono richiedono che la topologia della rete sia nota almeno ad alcuni dei suoi nodi. La trasmissione avviene in queste fasi:

1. Un nodo, che chiameremo sorgente, decide di inviare un pacchetto di dati ad un altro nodo, la destinazione.

2. Viene eseguito un algoritmo di ricerca che fornisce come output un percorso per il pacchetto che non sia soggetto ad interferenza, cioè non collida con altre trasmissioni.

3. Il nodo sorgente provvede ad instradare il pacchetto seguendo il cammino appena calcolato. Le indicazioni sul tragitto da seguire sono allegate al pacchetto stesso e vengono recepite dai nodi che lo riceveranno.

4. Il pacchetto giunge a destinazione.

4.1 Local Repair A^*

Una prima soluzione quasi-distribuita per il problema di *pathfinding* è quella di considerare le traiettorie degli agenti in modo totalmente indipendente, trascurando cioè la possibilità di interferenza tra nodi e calcolando i cammini minimi separatamente. Quando dovesse verificarsi una collisione, per riuscire a giungere fino a destinazione l'agente reitera A^* tenendo conto dell'impossibilità di attraversare il nodo occupato dall'agente con cui si è scontrato.

Questo metodo, chiamato Local Repair A^* , si comporta in modo insoddisfacente con grafi complicati, e può generare situazioni di *stallo* da gestire con degli espedienti per introdurre alterazioni e disturbi nel grafo. Nonostante questi inconvenienti, LR ha il vantaggio che ogni nodo non è tenuto a sapere nulla del movimento degli agenti attorno a lui; l'algoritmo, cioè, è distribuito ma non cooperativo. I nodi, infatti, agiscono in modo indipendente e cercano la strada più breve verso la meta senza concordare il percorso del pacchetto con gli altri per evitare interferenza. Sebbene agire singolarmente riduca drasticamente l'*overhead*, comporta però soluzioni peggiori di quelle che sarebbero possibili concordando il percorso tra i vari nodi.

Nel paragrago 5.1 verranno introdotti dei perfezionamenti a Local Repair A^* che mirano a gestire in modo intelligente l'eventualità di collisione tra pacchetti.

4.2 Cooperative A^*

Per migliorare il comportamento cooperativo di Local Repair, Cooperative A^* (CA*) aumenta l'informazione condivisa tra i vari agenti. Intuitivamente l'idea è quella di evitare le collisioni mettendo in comune l'informazione su quali nodi sono *impegnati*, cioè non transitabili, in determinati istanti di tempo. Questa informazione è sfruttata per pianificare dei cammini che non generino interferenza. Per illustrare CA^* e gli altri algoritmi presentati da Silver in [3], c'è bisogno di astrarre il grafo definito nella Sezione 1; in particolare l'esigenza è quella di creare un nuovo grafo in cui la topologia della rete è coniugata con la dimensione temporale. La Figura 4.1 illustra intuitivamente come in questo algoritmo la ricerca di un cammino minimo non si estenda solamente nelle dimensioni dello spazio, ma venga sviluppata anche con una profondità temporale.

Per tenere traccia della dinamica temporale della ricerca, mostriamo come si crea il nuovo grafo spazio-temporale $ST = (V_{ST}, E_{ST})$ a partire dal grafico $G = (V, A)$ della Sezione 1. Dato un nodo $x \in V$, esso viene *espanso* in un insieme di T_{max} sotto-nodi, dove T_{max} è la massima profondità temporale con quale vogliamo che si sviluppi la ricerca (si veda Figura 4.2, dove $T_{max} = 2$). In sostanza, ogni nodo del grafo originale genera un nuovo nodo per ogni slot, dal primo fino al T_{max}-imo. Per non complicare ulteriormente la notazione, andiamo ad indicare con un pedice lo slot associato al nodo, ad esempio

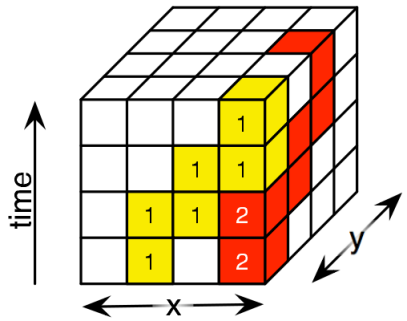


Figura 4.1: La nuova dimensione temporale del *pathfinding*

chiamiamo v_t il nodo che si è espanso da v ed è riferito all'istante t. Dal nodo originale $v \in V$, sono creati i sotto nodi $v_1, v_2, \ldots, v_{T_{max}}$, e l'insieme dei nodi di tutto il grafo è dato da:

$$V_{ST} = \{v_1, \ldots, v_{T_{max}}, \ \forall \ v \in E\} \tag{4.1}$$

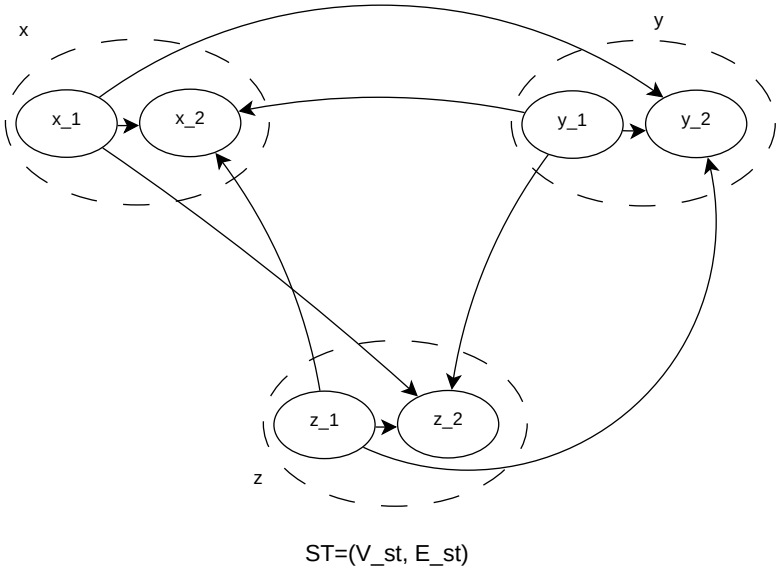


Figura 4.2: Il grafico $ST = (V_{st}, E_{st})$ con $T_{max} = 2$

Nel nuovo grafico ST, l'arco (v_t, x_p) viene creato solo se è permesso transitare da v a x in due istanti immediatamente successivi, cioè $(v_t, x_p) \in E_{ST}$ se e solo se:

1. $(v, x) \in E$ oppure $v = x$, cioè ci può essere comunicazione tra v e x,
2. $p = t + 1$, gli slot sono immediatamente successivi.

I costi dei nuovi archi sono assegnati sulla base del grafo originale:

$$c_{v_t x_p} = \begin{cases} c_{vx} & v \neq x \\ 0 & v = x \end{cases} \tag{4.2}$$

Dopo aver modificato il grafo di partenza in base alle nostre esigenze, vediamo come funziona CA^* . CA^* è un algoritmo di *pathfinding* basato su A^* , con la differenza che i cammini non vengono più ricercati sul grafico dei dispositivi originario, ma tutto ha luogo su $ST = (V_{ST}, E_{ST})$, da esso derivato. Uno alla volta gli agenti pianificano il loro percorso applicando A^* su ST e registrano i nodi che saranno visitati in una matrice M, chiamata *reservation table*. Se, ad esempio, all'istante t un agente ha in previsione di trovarsi nella posizione v, prenoterà per sé quello stato dello spazio-tempo segnando con un 1 l'elemento m_{vt} di M. In questo modo, per tutti gli agenti che calcoleranno il loro percorso successivamente, il passaggio per v_t sarà interdetto. Quando tutti gli agenti

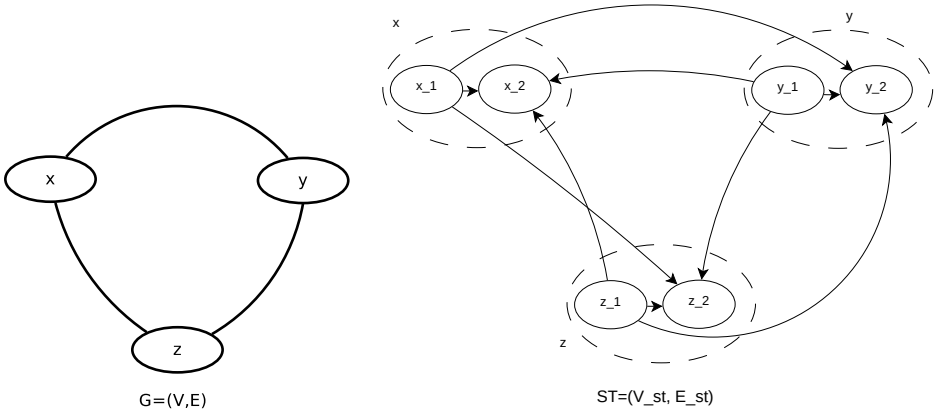


Figura 4.3: *Espansione* del grafo dei dispositivi in quello spazio-temporale

hanno pianificato, può iniziare la fase di esecuzione vera e propria, con la garanzia che non si verificheranno scontri.

Per gestire l'interferenza tra dispositivi all'interno di un raggio αd_{max}, si fa ricorso al grafo di interferenza introdotto nella Sezione 1. Quando si tratta di riservare degli stati spazio temporali nella *reservation table*, non viene posto uguale ad 1 un elemento soltanto nella matrice M, ma si segnano come non attraversabili anche tutti quelli stati che possono dare interferenza. Se, ad esempio, un pacchetto è inviato da x a v all'istante t, con $x \neq v$, considero non transitabili i nodi impegnati nella trasmissione, cioè:

$$\begin{cases} m_{vt} \longleftarrow 1 \\ m_{xt} \longleftarrow 1 \end{cases} \tag{4.3}$$

inoltre saranno inaccessibili tutti i nodi che interferiscono con x e v:

$$\begin{cases} m_{n,t} \longleftarrow 1 & \forall\, n \mid (v,n) \in E_i, \\ m_{m,t} \longleftarrow 1 & \forall\, m \mid (x,m) \in E_i, \end{cases} \tag{4.4}$$

dove E_i è l'insieme degli archi nel grafo di interferenza già definito. La gestione dell'interferenza è implementata nell'algoritmo CAMMINO() (Codice 4.2).

```
Algoritmo CA*(s, d)
    Closed ⟵ ∅
    Open ⟵ s_0
    g[s_0] ⟵ 0
    h[s_0] ⟵ MANHATTAN (s, d)
    f[s_0] ⟵ h[s_0]
    pred[s_0] ⟵ s_0
    while Open ≠ ∅
        k_t ⟵ argmin{f[i_j], i_j ∈ Open}
        if k = d
            return CAMMINO(pred[k_t])        /* Successo */
        Open ⟵ Open ∖ {k_t}
        Closed ⟵ Closed ∪ {k_t}
        for each x_p ∉ Closed | (k_t, x_p) ∈ E and m_xp = 0
            if x_p ∉ Open
              Open ⟵ Open ∪ {x_p}
              g[x_p] ⟵ g[k_t] + c_{x_p,k_t}
              h[x_p] ⟵ MANHATTAN((x, d)
              f[x_p] ⟵ g[x_p] + h[x_p]
              pred[x_p] ⟵ k_t
            else if (g[x_p] > g[k_t] + c_{x_p,k_t})
              g[x_p] ⟵ g[k_t] + c_{x_p,k_t}
              f[x_p] ⟵ g[x_p] + h[x_p]
              pred[x_p] ⟵ k_t
    return insuccesso /* Insuccesso */
```

Codice 4.1: Cooperative A^* .

Nel Codice 4.1 è contenuta una versione di CA^* in pseudo-codice. Si tratta di una modifica di A^* che cerca un cammino dal nodo s_0 aggiornando la matrice degli stati non consentiti. L'algoritmo è di tipo distribuito perchè ogni agente può eseguirlo in modo indipendente, tuttavia gli elementi m_{xy} della matrice devono poter essere letti ed aggiornati da tutti gli agenti.

L'algoritmo CAMMINO, presentato nel Codice 4.2, è ausiliario a CA^* e ricostruisce in modo ricorsivo il percorso dell'agente aggiornando di volta in volta la *reservation table* per impedire l'interferenza.

Pur migliorando il comportamento cooperativo, CA^* presenta alcuni inconvenienti: la soluzione ottenuta, oltre ad essere sub-ottima, dipende fortemente dall'ordine con cui gli agenti calcolano il loro cammino ed aggiornano la matrice; in secondo luogo, è possibile rintracciare alcune classi di problemi per cui l'esecuzione non è possibile in modo cooperativo.

```
Algoritmo CAMMINO(v_t)
    if pred[v_t] = v_t
        return v_t
    x_p ⟵ pred[v_t]
    if x ≠ v
        m_xt ⟵ 1
        m_vt ⟵ 1
        for each n|(v,n) ∈ E_i /*non traversabili i nodi*/
            m_n,t ⟵ 1 /*per cui ho interferenza*/
    return {v_t} ∪ CAMMINO(pred[v_t])
```

Codice 4.2: L'algoritmo ausiliario CAMMINO A^*

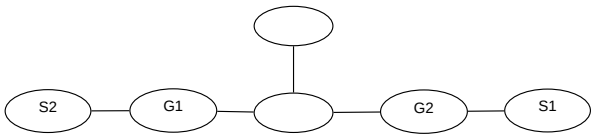


Figura 4.4: Esempio di problema che non ha soluzione con l'algoritmo CA*

In questi casi una soluzione *vorace* per un agente preclude agli altri il raggiungimento della meta. Qualora si verificasse una situazione del genere, il tempo impiegato per il routing è pari alla somma dei tempi necessari ad eseguire ogni richiesta separatamente. Si consideri ad esempio la rete di Figura 4.4, in cui abbiamo due richieste, la prima da $s1$ a $g1$ e la seconda da $s2$ a $g2$. Poiché i due cammini sono calcolati separatamente, l'agente che pianifica per primo sceglierà un percorso rettilineo che lo porti direttamente all'obbiettivo; in questo modo il secondo agente è bloccato e non potrà raggiungere la meta, o meglio, dovrà aspettare che la prima richiesta giunga al termine per iniziare a muoversi. E' proprio per ovviare a questo inconveniente che nella definizione del grafico ST dobbiamo inserire degli archi del tipo (v_t, v_{t+1}) (si veda la Figura 4.2 e la procedura costruzione degli archi), che consentono all'agente di restare fermo durante il proprio turno. In questo modo, il pacchetto non viene inviato durante uno slot, ma aspetta l'istante successivo rimanendo nella memoria del dispositivo che lo aveva ricevuto. Nell'implementazione dell'algoritmo CAMMINO(), inoltre, la matrice M non viene aggiornata se non c'è una effettiva trasmissione tra due dispositivi diversi; ciò permette a due agenti di essere nello steso nodo contemporaneamente. Con questi accorgimenti, il caso irrisolvibile in Figura 4.4 è gestibile: per due slot consecutivi i pacchetti avanzano verso il nodo centrale, dove saranno immagazzinati entrambi nella memoria del dispositivo; negli slot successivi vengono trasmessi uno per volta e posono giungere a destinazione.

CA* ha lo svantaggio di imporre una netta distinzione tra la fase di *planning* e quella di *routing*, cioè tra la pianificazione dei cammini e la loro effettiva esecuzione. In un contesto di reti *wireless* questa deve considerarsi come una limitazione, infatti la composizione della rete potrebbe subire delle variazioni durante la fase di planning, e i cammini calcolati

perderebbero qualsiasi utilità.

4.3 Hierarchical Cooperative A^*

Introducendo A^* , è emerso come la velocità di esecuzione dell'algoritmo sia drasticamente influenzata dall'euristica scelta; Cooperative A^* , ad esempio, paga in termini di prestazioni le scarse *performance* di Manhattan, non sempre competitiva per grafici complessi. Nel Codice 4.3 è presentata una versione modificata di CA^* chiamata *Hierarchical Cooperative A**, essa fa uso dell'euristica *Reverse Resumable A**, più sofisticata rispetto a Manhattan (vedi Sezione 2).

```
Algoritmo HCA*(s, d)
    INITIALISERRA*(s, d)
    Closed ⟵ ∅
    Open ⟵ s_0
    g[s_0] ⟵ 0
    h[s_0] ⟵ ABSTRACTDIST (s, d)
    f[s_0] ⟵ h[s_0]
    pred[s_0] ⟵ s_0
    while Open ≠ ∅
        k_t ⟵ argmin{f[i_j], i_j ∈ Open}
        if k = d
            return CAMMINO(pred[k_t])        /* Successo */
        Open ⟵ Open ∖ {k_t}
        Closed ⟵ Closed ∪ {k_t}
        for each x_p ∉ Closed | (k_t, x_p) ∈ E_ST and m_xp = 0
            if x_p ∉ Open
                Open ⟵ Open ∪ {x_p}
                g[x_p] ⟵ g[k_t] + c_{x_p,k_t}
                h[x_p] ⟵ ABSTRACTDIST((x, d)
                f[x_p] ⟵ g[x_p] + h[x_p]
                pred[x_p] ⟵ k_t
            else if (g[x_p] > g[k_t] + c_{x_p,k_t})
                g[x_p] ⟵ g[k_t] + c_{x_p,k_t}
                f[x_p] ⟵ g[x_p] + h[x_p]
                pred[x_p] ⟵ k_t
    return insuccesso /* Insuccesso */
```

Codice 4.3: Hierarchical Cooperative A^*

Reverse Resumable A^* , è un'euristica che sfrutta A^* stesso per stimare la distanza di un nodo dalla destinazione, a differenza dell'algoritmo di ricerca, però, RRA^* non considera le possibili interferenze tra agenti e procede a ritroso dalla destinazione fino al nodo in eleborazione. Questa soluzione è vantaggiosa perchè:

- La stima è molto accurata, infatti, sempre rimanendo un limite inferiore per la distanza reale, vi si avvicina di più rispetto a Manhattan.

- La scelta di procedere a ritroso permette di riutilizzare le distanze già stimate: una volta che un nodo è stato inserito nella lista di quelli già elaborati (C), abbiamo la sicurezza che il costo del cammino che lo collega al nodo iniziale sia minimo. Partire dalla fine, quindi, ci permette di riutilizzare le euristiche già calcolate senza eseguire l'algoritmo.

```
Algoritmo ABSTRACTDIST(i, d)
    if i ∈ C
        return g'[i]
    if RESUMERRA*(i) = successo
        return g'[i]
    return +∞

Algoritmo INITIALISERRA*(s, d)
    g'[d] ⟵ 0
    h'[d] ⟵ MANHATTAN(d, s)
    O ⟵ d /*Lista dei nodi che RRA* deve processare*/
    C ⟵ ∅ /*Lista dei nodi processati da RRA* */
    RESUMERRA*(s)

Algoritmo RESUMERRA*(i)
    while O ≠ ∅
        k ⟵ argmin{f[j], j ∈ O}
        C ⟵ C ∪ {k}
        O ⟵ O ∖ {k}
        if i = k
            return successo
        for each j ∉ C | c_kj < +∞
            if j ∉ O
                O ⟵ O ∪ {j}
                g'[j] ⟵ g'[k] + c_kj
                h'[j] ⟵ MANHATTAN(j, s)
                f'[j] ⟵ g'[k] + h'[j]
            else if g'[j] < g'[k] + c_kj]
                g'[j] ⟵ g'[k] + c_kj
                f'[j] ⟵ g'[j] + h'[j]
                pred[j] ⟵ k
    return insuccesso
```

Codice 4.4: Reverse Resumable A^*

In Codice 4.4 si trova una descrizione dell'euristica Reverse Resumable A*. Riassumiamo le funzioni associate ad ogni nodo:

f, g, h sono le funzioni di costo e di euristica che abbiamo già visto nella spiegazione dell'algoritmo A*;

f', g', h' sono altre tre funzioni, del tutto analoghe alle precedenti, che però sono utilizzate nell'euristica, che, come abbiamo visto è essa stessa basata sull'algoritmo A*.

Per tenere traccia correttamente delle distanze già stimate, è necessario memorizzare per ogni nodo trattato le liste dei nodi elaborati e da elaborare, C ed O, e il valore delle funzioni g', f', h'.

I vantaggi introdotti da HCA* riguardano soprattutto il tempo computazionale. Avere una buona euristica, infatti, permette all'algoritmo A* di elaborare meno nodi e giungere alla soluzione con un numero inferiore di iterazioni del ciclo principale (vedi Codice 2.1).

4.4 Windowed Hierarchical Cooperative A^*

Gli algoritmi già visti introducono alcuni miglioramenti nel tempo di calcolo e nella qualità delle soluzioni trovate, tuttavia presentano ancora dei problemi:

- la soluzione ottenuta è fortemente influenzata dall'ordine con cui gli agenti creano il percorso;
- le condizioni del grafo potrebbero variare mentre un agente sta già seguendo il cammino che ha calcolato: in tal caso l'intero algoritmo deve essere eseguito da capo;
- la fase di planning e di esecuzione del cammino sono nettamente divise;
- ogni nodo necessita di scambiare con gli altri un quantitativo di informazione abbastanza consistente (si veda la Sezione 4.5).

Per risolvere alcuni di questi inconvenienti, [3] propone di suddividere la ricerca in tante finestre temporali: ogni agente cerca una cammino parziale che sia ottimo per w slot, inizia a percorrerlo e dopo w passi ricalcola l'algoritmo dal punto in cui è arrivato. Questa strategia riduce la dipendenza della soluzione globale dell'ordine con cui gli agenti scelgono il loro percorso, rendendo lo scenario più dinamico. Per assicurarsi che, seppure incompleto, il cammino si snodi nella direzione corretta, la ricerca cooperativa è limitata alla profondità temporale $w < T_{max}$, mentre la parte rimanente fino alla destinazione è calcolata senza tenere in conto gli altri agenti. Per implementare questa soluzione si usa un espediente: per ogni nodo v_w, raggiunto dopo w passi, è inserito nel grafo un arco fittizio che collega v_w direttamente alla destinazione d, con un costo pari alla distanza euristica.

Nel Codice 4.5 realizziamo questo algoritmo modificando HCA^* in modo che per ogni nodo v_w venga creato un arco fittizio che unisce v_w direttamente con $goal_{w+1}$. E' evidente che il cammino trovato consterà al più di $w+1$ archi, ma l'ultimo non verrà mai percorso: dopo che l'agente si è mosso per w volte, infatti, l'esecuzione si blocca e il cammino dovrà essere ricalcolato.

```
Algoritmo WHCA*(s, d)
    INITIALISERRA*(s, d)
    for each x ∈ E /*Per ogni nodo al tempo w*/
        E_ST ⟵ E_ST ∪ {(d_{w+1}, x_w)}/*aggiungo al grafo lato fittizio*/
        c_{x_w d_{w+1}} ⟵ ABSTRACTDIST(x,d) /*e assegno un costo*/
    Closed ⟵ ∅
    Open ⟵ s_0
    g[s_0] ⟵ 0
    h[s_0] ⟵ ABSTRACTDIST (s, d)
    f[s_0] ⟵ h[s_0]
    pred[s_0] ⟵ s_0
    while Open ≠ ∅
        k_t ⟵ argmin{f[i_j], i_j ∈ Open}
        if k = d
            return CAMMINO(pred[k_t])        /*Successo*/
        Open ⟵ Open ∖ {k_t}
        Closed ⟵ Closed ∪ {k_t}
        for each x_p ∉ Closed | (k_t, x_p) ∈ E_ST and m_xp = 0
            if x_p ∉ Open
                Open ⟵ Open ∪ {x_p}
                g[x_p] ⟵ g[k_t] + c_{x_p,k_t}
                h[x_p] ⟵ ABSTRACTDIST((x, d)
                f[x_p] ⟵ g[x_p] + h[x_p]
                pred[x_p] ⟵ k_t
            else if (g[x_p] > g[k_t] + c_{x_p,k_t})
                g[x_p] ⟵ g[k_t] + c_{x_p,k_t}
                f[x_p] ⟵ g[x_p] + h[x_p]
                pred[x_p] ⟵ k_t
    return insuccesso /*Insuccesso*/
```

Codice 4.5: Windowed Hierarchical Cooperative A^*

4.5 Problematiche realizzative

Pur non richiedendo che tutti i cammini vengano calcolati insieme, le soluzioni viste in questa sezione impongono che ogni agente tenga traccia del comportamento degli altri, aggiornando continuamente la matrice M. Se una soluzione di questo tipo è ammissibile nelle applicazioni informatiche, in un ambito di reti *ad hoc* comporterebbe alcune difficoltà realizzative, infatti dovremmo ammettere la presenza di una *unità centrale* in grado di coordinare i vari nodi scambiando informazioni sui rispettivi cammini. Questo richiederebbe per ogni dispositivo un canale di comunicazione molto affidabile e con ampia banda con l'unità centrale, e produrrebbe quindi un'implementazione estremamente costosa e poco adatta a reti locali.

Una migliore alternativa potrebbe essere quella di ipotizzare che solo i nodi sorgente comunichino tra loro: una volta che hanno *concordato* un cammino tale da evitare l'interferenza, possono allegare al pacchetto da trasmettere l'informazione relativa al corretto percorso che questo deve seguire nella rete. Tali indicazioni saranno recepite e attuate

dai nodi che formano il cammino. Una soluzione di questo tipo, sebbene di più facile implementazione, renderebbe impossibile cambiare percorso *in itinere* in seguito ad eventi inaspettati, e nemmeno l'algoritmo WHCA^* sarebbe più applicabile: infatti se dopo w passi il pacchetto si trovasse in un nodo che non è nell'insieme delle sorgenti, non avrebbe le informazioni necessarie a pianificare il suo cammino.

Tutti gli algoritmi visti fin ora, poi, hanno l'inconveniente di esigere la completa conoscenza della rete almeno da parte di un nodo, e di imporre che i vari dispositivi si scambino informazioni relative ai rispettivi tragitti (la matrice M). Non sempre questi vincoli possono essere soddisfatti, specialmente in reti *ad hoc*, dove la topologia di una rete può essere dinamica. Nella Tabella 4.1 vengono confrontati gli algoritmi quasi-distribuiti in base ai dati che ogni agente è tenuto a conoscere e a scambiare con gli altri, viene data anche una stima molto approssimativa del contenuto informativo associato.

Algoritmo	**Conoscenza**	**Dati scambiati**	**Info scambiata**
Local Repair	Grafo e costi	Nessuno	0 bit
CA^*	Grafo e costi	M $\in \mathbb{N}^{n \times m}$	$\lvert V \rvert \cdot T_{max}$ bit
HCA^*	Grafo e costi	M $\in \mathbb{N}^{n \times m}$	$\lvert V \rvert \cdot T_{max}$ bit
WCA^*	Grafo e costi	M $\in \mathbb{N}^{n \times m}$	$\lvert V \rvert \cdot T_{max}$ bit

Tabella 4.1: Confronto tra gli algoritmi distribuiti. V è l'insieme dei vertici e T_{max} la massima profondità temporale con cui si estende la ricerca.

Capitolo 5

Approccio distribuito

Come già emerso in precedenza, gli algoritmi quasi-distribuiti hanno il pregio di pianificare un routing sub-ottimo ma privo di qualsiasi interferenza tra dispositivi. Questa caratteristica è pagata in termini di una onerosa comunicazione tra i vari nodi (vedi Sezione 4.5), che devono scambiarsi informazioni sui rispettivi cammini per evitarne le intersezioni. In questa sezione presentiamo alcuni approcci che, allentando i vincoli di ottimalità dei cammini trovati, riescano a ridurre l'informazione condivisa tra i vari nodi.

5.1 LRA* con gestione delle collisioni

Analizzando l'algoritmo Local Repair A^* (Sezione 4.1), non ci siamo soffermati sui vantaggi che indubbiamente una soluzione di quel tipo comporta:

- minima separazione tra la fase di *planning* e l'esecuzione del cammino,
- minimo *overhead* (vedi Tabella 4.1).

Lo svantaggio maggiore di LRA^* sta nella gestione delle collisioni: appare poco sensato ricalcolare da capo A^* ogni qualvolta se ne presenti una, inoltre non è sempre facile individuare con certezza tutte le interferenze che si sono verificate all'interno della rete. Si rischierebbe di non gestire alcuni casi di trasmissione fallita, impedendo ai pacchetti di giungere a destinazione. Preso atto delle criticità di LRA*, vediamo come migliorarne il comportamento.

Per evitare di dover calcolare nuovamente il cammino del pacchetto ad ogni collisione, possiamo abbinare LRA* al metodo di gestione delle collisioni CSMA/CA, adottato anche nello standard IEEE 802.11. Diamo ora una descrizione qualitativa di questo metodo, rimandando a [4] per una trattazione più dettagliata.

CSMA

CSMA (Carrier Sense Multiple Access) è una strategia che permette a più dispositivi di utilizzare lo stesso canale ed in particolare la stessa banda di frequenze per trasmettere pacchetti di dati. L'idea è che prima di ogni trasmissione il dispositivo ascolti il canale per verificarne lo stato: se il mezzo è libero procede a inviare il pacchetto, altrimenti aspetta un po' e ascolta nuovamente il canale. E' importante che il dispositivo in attesa aspetti un tempo casuale prima di riprovare, altrimenti c'è il pericolo che tutti i nodi trasmettano contemporaneamente non appena il canale è tornato libero. Il protocollo *p-persistent CSMA* adatta questa tecnica anche a sistemi di comunicazione in cui il tempo è suddiviso in slot temporali: quando un dispositivo trova il canale libero trasmette il pacchetto con probabilità p, altrimenti differisce la trasmissione allo slot immediatamente successivo con probabilità $1 - p$.

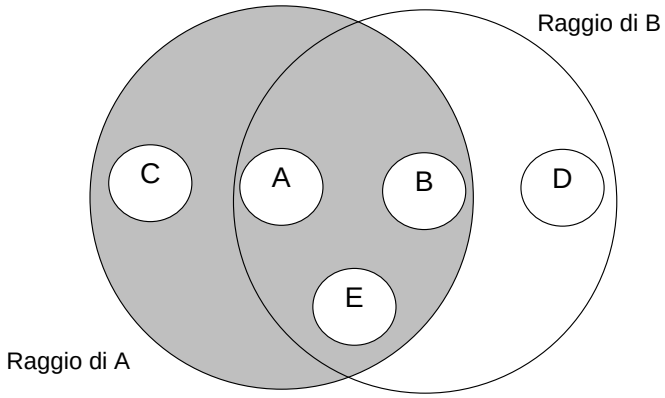


Figura 5.1: Il problema del terminale nascosto

CSMA/CA

Nella forma in cui lo abbiamo presentato, per il protocollo CSMA sussiste il problema del *terminale nascosto*. Consideriamo lo scenario di Figura 5.1: immaginiamo che A stia trasmettendo un pacchetto a B; anche il dispositivo D vuole trasmettere a B, vediamo cosa succederebbe se D usasse il protocollo CSMA: per prima cosa ascolterebbe il canale, essendo fuori dal raggio di A lo troverebbe libero e inizierebbe a trasmettere creando interferenza con B, che si trova nel raggio sia di A che di D. La trasmissione fallisce nonostante D abbia ascoltato il canale prima di procedere. In ambienti *wireless*, infatti, per evitare le collisioni devo ascoltare il canale al ricevitore, non al trasmettitore.

Per risolvere lo sgradevole problema del terminale nascosto, è stato implementato il protocollo CSMA/CA. Ipotizziamo che il nodo A debba inviare un pacchetto a B, per prima cosa manda un RTS (Request To Send), che è un pacchetto molto corto contenente la lunghezza dei dati che A desidera inviare. Ricevuto l'RTS, il nodo B può decidere di

accordare il permesso e risponde con un CTS (Clear To Send). Quando la trasmissione è avvenuta, B conferma la riuscita della comunicazione con un pacchetto ACK. Consideriamo ora la trasmissione dal punto di vista dei nodi C e D, che non ne sono coinvolti: C è nel raggio di A e riceve l'RTS, per non causare interferenza decide di restare inattivo fino a che lo scambio si è concluso. Poiché nell'RTS è contenuta la misura della sequenza da inviare, C può stimare per quanto tempo dovrà tacere, e considerare il canale virtualmente occupato in quel periodo allocando un NAT (Network Allocation Vector). Il NAT ha la funzione di un promemoria, ricorda al nodo che lo alloca di non accettare ne' richiedere connessioni. Il nodo D si trova nel raggio di B ma non in quello di A, quindi allocherà il suo NAT non appena riceve il pacchetto CTS da B.

Usando il protocollo CSMA/CA, siamo in grado di mandare un pacchetto attraverso una rete di dispositivi seguendo un cammino calcolato con l'algoritmo LRA* e senza dover ricalcolare tutto il percorso in seguito ad una collisione. Il limite di questa soluzione è che globalmente si assiste ad un comportamento non cooperativo tra i nodi: ognuno instrada i suoi pacchetti senza preoccuparsi di quello che sta succedendo intorno a sé e di come si potrebbe minimizzare il costo totale dei cammini. D'altro canto la comunicazione ausiliaria tra nodi è ridotta al minimo indispensabile ed è prevista solo tra dispositivi adiacenti. Nel resto della sezione vedremo un altro metodo che permette di aumentare la cooperazione tra i nodi senza richiedere un aumento dell'*overhead*.

5.2 Paradigma *Agent Centered*

Nel lavoro [5], Sven Koenig introduce un nuovo approccio al *pathfinding* che lui stesso definisce *agent-centered*, basato cioè sulla centralità dell'agente che sta compiendo la ricerca e sullo spazio a lui circostante. Questo paradigma differisce dagli altri perchè il planning e l'esecuzione del cammino sono continuamente alternati. Un aspetto estremamente interessante è che la ricerca agisce in modo distribuito, avviene cioè nella porzione di grafo adiacente alla posizione del pacchetto. Per questa ragione ai nodi non è richiesta la conoscenza completa della topologia della rete; basta solo che riescano a stimare in modo euristico la loro distanza rispetto agli altri. Nella sua trattazione, Koenig descrive l'approccio agent-centered per un solo agente in movimento, cioè per una singola richiesta; per rendere la spiegazione più semplice ci atteniamo a questa descrizione, andremo poi ad ampliarla per gestire il caso cooperativo. Iniziamo elencando le ipotesi usate:

- la ricerca avviene nel solito grafico $G = (V, E)$ (Sezione 1),
- i nodi non sono tenuti a conoscere interamente la topologia della rete,
- ogni nodo è in grado formulare un'euristica, cioè, se non conosce la distanza esatta verso un altro nodo, può stimarla per difetto,

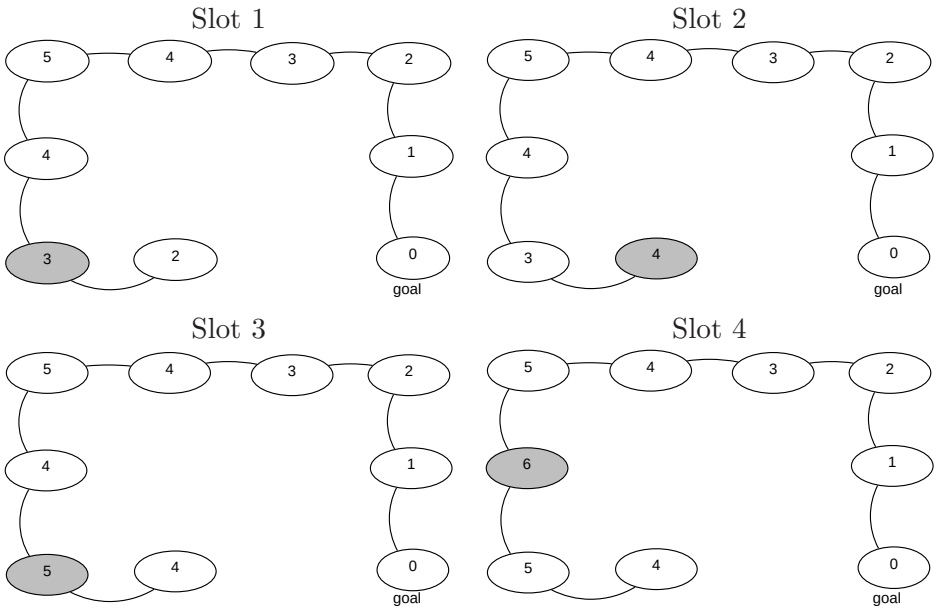


Figura 5.2: Esempio di algoritmo LRTA*. I nodi colorati di grigio sono quelli che ricevono il pacchetto nello slot corrente.

- ogni nodo sa quali e quanti nodi sono immediatamente adiacenti a lui, e può richiedere le loro euristiche.

Presentiamo ora un algoritmo di tipo agent-centered chiamato Learning Real Time A*. LRTA* alterna planning ed esecuzione e compie ad ogni passo la scelta migliore in relazioni alle informazioni locali di cui ogni nodo dispone. Si consideri la rete elementare in Figura 5.2, ogni nodo è associato ad un valore (nella figura rappresentato all'interno) che è la distanza euristica che separa la sua posizione dall'obiettivo (goal). Questo valore, unitamente al costo del lato, è l'unico elemento usato dall'agente per scegliere la direzione verso cui muoversi. Ovviamente tutti i nodi possono essere possibili destinazioni per i pacchetti, quindi teniamo presente che ogni dispositivo dovrà conservare una distanza euristica per ogni possibile destinazione (sono $N - 1$). Per semplicità, noi ci riferiremo all'euristica associata ad un nodo intendendo quella relativa alla destinazione del pacchetto che deve essere trasmesso. In conformità con le ipotesi, nodi adiacenti hanno accesso alle rispettive euristiche attraverso il campo *heuristic*[ ].

Poiché abbiamo associato un'euristica ad ogni nodo, potremmo pensare che la mossa migliore per l'agente sia quella di muoversi sempre verso il nodo più vicino alla meta, cioè verso quel nodo che tra tutti minimizza la somma tra c_{ab} e il valore dell'euristica. Questo è vero solo in parte: infatti l'agente potrebbe trovarsi *imprigionato* in una regione di minimo locale, una zona in cui tutti i nodi adiacenti hanno euristiche maggiori di quella corrente. Per uscire da questa situazione di *stallo*, è necessario tenere costantemente aggiornati i valori euristici in modo che siano i più vicini possibile alle distanze reali. Per

una rappresentazione intuitiva si veda la Figura 5.3, in cui l'altezza di un nodo corrisponde alla distanza euristica rispetto alla meta: l'agente tende a *cadere* verso le zone a minor quota (che dovrebbero essere più prossime alla meta), tuttavia può trovarsi bloccato in una *valle*, per cui ha bisogno di aumentare il suo potenziale per riprendere la discesa. Detto in modo più formale, la valle corrisponde ad una euristica troppo bassa rispetto al valore reale della distanza fino alla meta; in tal caso c'è bisogno di ricalcolarla affinché sia più accurata.

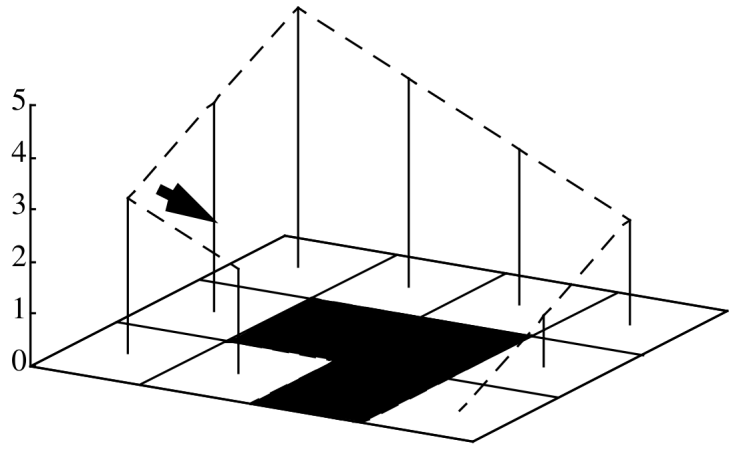


Figura 5.3: Superficie dei valori associati ai nodi

Spieghiamo con un esempio (Figura 5.2) l'algoritmo LRTA*, dove per semplicità i costi degli archi sono considerati unitari: dalla posizione iniziale l'agente si sposta nel nodo alla sua destra perché è quello con euristica minore (il costo dell'arco vale uno), dopodiché ha bisogno di aggiornare il valore associato a quel nodo, altrimenti non potrebbe più proseguire perché si trova in una zona di minimo locale. L'aggiornamento avviene sommando 1 (costo del lato percorso) al valore associato al nodo di provenienza. La seconda mossa riporta l'agente alla posizione iniziale, e il valore euristico memorizzato passa da 3 a 5. Poi si muove verso il nodo sopra di lui perché ha valore minore rispetto ai nodi adiacenti, infine aggiorna l'euristica come prima. A questo punto tutti i valori sono decrescenti e la trasmissione arriva direttamente a destinazione. L'esecuzione viene completata in 9 passi. Nel Codice 5.1 è illustrato in maniera più formale l'algoritmo eseguito ad ogni mossa.

Un aspetto interessante di questo algoritmo è che successive esecuzioni della stessa richiesta vengono eseguite con costi via via inferiori: intuitivamente possiamo dire che i nodi *imparano* dai propri errori. Questo comportamento è dovuto ai successivi raffinamenti della stima euristica, che dopo vari aggiustamenti arriva ad essere un *lower bound* molto stretto della distanza reale. Se, ad esempio, ricalcolassimo il cammino del nostro esempio, otterremmo ancora una soluzione in 9 mosse, ma eseguendolo per la terza volta,

```
Algoritmo NEXT(d)
    a ⟵ CURRENTPOSITION()
    b ⟵ argmin{heuristic[i] + c_ai, ∀ i| (a, i) ∈ E}
    MOVETO(b)
    if b = d
        return
    if   ∀ i|(b, i) ∈ E : c_bi + heuristic[i] = cost
        heuristic[b]⟵ heuristic[a] +c_ab
    return
```

Codice 5.1: Esecuzione di una mossa in LRTA^*

ne basterebbero solo 7.

L'algoritmo LRTA* risolve il problema del *pathfinding* in maniera distribuita, richiede una conoscenza limitata del grafo da parte dei suoi nodi e agisce in real-time. Tuttavia è una soluzione che non gestisce l'interferenza e non agisce in modo cooperativo. Nella prossima sezione presenteremo un lavoro che cerca di integrare l'approccio agent-centered con queste due esigenze.

5.3 *Flocks* e *Destination Map*

Jansen e Sturtevant ravvisano alcuni limiti negli algoritmi cooperativi proposti da Silver in [3]: innanzitutto sono troppo onerosi in termini di informazione condivisa tra agenti, non sfruttano appieno la cooperazione per scambiare informazioni sulla dinamica dei cammini e difficilmente posono essere applicati a quelle situazioni in cui un nodo abbia informazione incompleta sul resto della rete. In [6] vengono elaborati dei metodi per rendere cooperativo l'approccio agent-centered di Koenig; per farlo si prova ad adattare l'idea di *flock* [7] al problema del *pathfinding* cooperativo. La parola *flock*, ossia stormo, indica una serie di algoritmi di *pathfinding* che si ispirano al movimento degli animali in natura: un gruppo di uccelli, infatti, riesce a coordinare il volo di tutti i suoi componenti (senza che si scontrino) seguendo una rotta condivisa da tutti, formando cioè uno stormo. Nel nostro scenario, come per gli animali, quanto più un agente si muove in armonia con gli altri, tanto minori saranno le collisioni.

Un altro sistema naturale che ha fornito elementi utili per l'algoritmo di Jansen e Sturtevant è il formicaio: le formiche, oltre a muoversi in schiere ordinate, lasciano sul terreno dei componenti chimici come il *feromone* per comunicare il percorso verso il cibo alle altre formiche che verranno dopo di loro. In *Ant System* di Dorigo *et al.* [8], è sviluppato un metodo di ricerca basato sull'idea che sia conveniente lasciare delle traccie sul grafo per condividere un'informazione dinamica che permetta di riutilizzare i cammini già cercati.

Con l'intento di implementare un algoritmo che realizzi l'idea di flock, in [6] viene introdotta una nuova struttura dati distribuita chiamata *direction map* (DM) che contiene

l'informazione sulle direzioni con cui i pacchetti si sono mossi all'interno del grafo negli slot precedenti. La DM è usata in fase di *planning* per scoraggiare gli agenti dal seguire direzioni differenti da quelle precedentemente utilizzate, si cerca, cioè, di spostare gli agenti in modo coordinato. Nel loro lavoro, Jansen e Sturtevant realizzano una *Direction Map* associando ad ogni nodo un vettore geometrico che rappresenta la media pesata delle direzioni con cui gli agenti sono usciti da quel nodo precedentemente. L'idea di base è quella di tenere traccia dei movimenti degli agenti nel grafo e avere per ogni dispositivo una statistica delle direzioni *preferite*. Purtroppo, però, il grafo dei dispositivi del Capitolo 1 rappresenta una astrazione della rete reale di terminali *wireless*, cioè non ha immediato significato geometrico. Abbiamo visto infatti che un singolo nodo della rete dei dispositivi potrebbe corrispondere ad un insieme di nodi della rete reale, di conseguenza gli archi non sono correlati alle direzioni con cui effettivamente vengono spediti i pacchetti. Per questo motivo sarebbe difficile applicare direttamente i metodi proposti in [6] al nostro problema. Tuttavia l'idea di tenere una statistica dei movimenti dei pacchetti in modo da dirigerli in direzioni preferenziali é valida e per questo proveremo ad adattarla al nostro scenario.

Il modo più semplice per implementare la *direction map* senza legarci agli aspetti geometrici è quello di dare un *punteggio* ad ogni nodo della rete. Cercheremo di incentivare gli agenti a spostarsi verso i nodi con punteggio maggiore. In fase di inizializzazione, per ogni nodo vicino a inizializza un contatore che rappresenta il punteggio relativo a quel nodo.

$$a.count_i \longleftarrow 1 \quad \forall\, i \mid (a,i) \in E \tag{5.1}$$

Ogni volta che a spedisce un pacchetto verso il dispositivo b, incrementa di un'unità il punteggio di quel nodo.

$$a.count_b + + \tag{5.2}$$

I contatori sono utilizzati in fase di *planning*: intuitivamente, se un agente cerca di muoversi verso un nodo che ha un contatore più basso degli altri, dovrà pagare un costo extra. Dati i nodi a e b, il costo del lato (a,b) è ottenuto dividendo c_{ab} per un termine α. Il fattore α non è altro che il punteggio di b normalizzato rispetto alla media di tutti i punteggi dei nodi adiacenti ad a. Se, ad esempio, ci sono K nodi vicini ad a, allora il costo EDGECOST(a,b) del lato (a,b) è dato:

$$EDGECOST(a,b) = \frac{c_{ab}}{\alpha}\ , \quad \text{con } \alpha = \frac{a.count_b}{\frac{1}{K}\sum_{i=1}^{K} a.count_i} \tag{5.3}$$

In questo modo, un agente sarà disincentivato a muoversi verso un nodo che ha punteggio molto inferiore rispetto agli altri. Il nodo a cui trasmettere il pacchetto viene scelto in modo da minimizzare la somma di EDGECOST() e dell'euristica di quel nodo:

$$b \longleftarrow \ argmin\{heuristic[i] + \text{EDGECOST(a, i)}, \forall\, i | (a,i) \in E\} \tag{5.4}$$

Macroscopicamente, ci attendiamo che i pacchetti assumano un comportamento più ordinato rispetto ad algoritmi come LRA* in cui ogni agente si muove in modo totalmente indipendente dagli altri. Nel Codice 5.2 è riportata la procedura di calcolo del costo di un arco.

```
Algoritmo EDGECOST( a ,  b )
    K ⟵ a.NUM_VICINI()
    for each i | (a, i) ∈ E
        α ⟵ α + a.count_i
    α ⟵ α/K
    return c_ab/α
```

Codice 5.2: Calcolo del costo di un arco

A questo punto vogliamo realizzare un'integrazione tra l'algoritmo LRTA* e la *direction map* che abbiamo costruito: i nodi spediranno i pacchetti secondo la politica *vorace* di LRTA*; la presenza della direction map servirà a favorire lo sviluppo di un flusso ordinato di pacchetti con un numero esiguo di collisioni. L'influenza dei contatori sulle scelte real-time di LRTA* deriva dal fatto che il costo di ogni lato è calcolato con la procedura EDGECOST(). Nel Codice 5.3 è implementata una modifica dell'algoritmo LRTA* che gestisce la *direction map* aggiornando i punteggi ogni volta ha luogo una transizione, e usando la procedura EDGECOST() per calcolare i costi degli archi.

```
Algoritmo NEXT( d )
    a ⟵ CURRENTPOSITION()
    b ⟵ argmin{heuristic[i] + EDGECOST(a, i), ∀ i|(a, i) ∈ E}
    MOVETO( b )
    while INTERFERENZA( b ) = true and  a.Vicini≠ ∅
            b ⟵ argmin{heuristic[i] + EDGECOST(a, i), ∀ i|(a, i) ∈ a.Vicini}
            MOVETO( b )
            a.Vicini ⟵ a.Vicini ∖ {b}
    if a.Vicini= ∅
        WAIT ( )
        return
    a.count_b + +
    if b = d
        return
    if  ∀ i|(b, i) ∈ E: EDGECOST(b, i) +heuristic[i] = cost
        heuristic[b]⟵ heuristic[a] +c_ab
    return
```

Codice 5.3: Esecuzione di una mossa in LRTA^*

L'uso delle DM insieme al paradigma agent-centered introduce una cooperazione tra agenti che dovrebbe convogliare i pacchetti in *flocks* in modo da limitare la frequenza di collisioni a livello statistico. Le collisioni, tuttavia, possono essere ridotte ma difficilmente saranno eliminate del tutto; è opportuno dotarsi di strumenti che ci permettano di non

perdere i dati quando queste si presentino. Una soluzione potrebbe essere quella di provare a inviare il pacchetto verso il nodo migliore secondo 5.4 e, in caso di collisione, escludere il nodo per cui si è verificata e reiterare il criterio di decisione 5.4. In altre parole, si invia il pacchetto al nodo che, tra quelli che non creano interferenza, minimizza la somma dell'euristica e di EDGECOST(). Riepiloghiamo i vantaggi dell'integrazione tra LRTA* e la direction map:

- i nodi non sono tenuti a conoscere interamente la topologia grafo, ma solo le posizioni dei nodi adiacenti,
- la ricerca è real-time e si adatta meglio a gestire situazioni dinamiche e imprevisti,
- la comunicazione tra nodi è ridotta e riguarda solo dispositivi adiacenti,
- le collisione sono minimizzate.

Tuttavia sono presenti anche svariati inconvenienti:

- determinare un'euristica ammissibile per le distanze richiede una fase preliminare di inizializzazione della rete,
- le soluzioni trovate sono qualitativamente peggiori rispetto agli algoritmi visti in precedenza,
- è necessario gestire individualmente le collisioni che non sono state eliminate,
- ogni nodo deve poter conservare in memoria $N-1$ valori euristici e un contatore per ogni nodo adiacente

Capitolo 6

Conclusioni

Dopo aver analizzato varie soluzioni per il problema che ci eravamo posti inizialmente, cerchiamo di capire quali sono i i punti di forza e le criticità dei tre diversi approcci presentati in questo lavoro: centralizzato, quasi centralizzato e distribuito.

Il metodo centralizzato del Capitolo 3, cioè il modello di *Programmazione Lineare Intera* e i *Vettori di Stato*, ha l'innegabile vantaggio di produrre soluzioni ottime, porta cioè i pacchetti a destinazione minimizzando il costo totale dei cammini. Questo si traduce in un risparmio in termini di potenza spesa e numero di *slot* impegnati nella trasmissione. In aggiunta questo approccio elimina completamente la presenza di collisioni, alleggerendo l'*overhead* imposto da una gestione CSMA/CA. Una soluzione di questo tipo ha lo svantaggio di adattarsi poco alle reti realmente *ad hoc*, cioè quelle in cui è totalmente assente una struttura di supporto ai dispositivi. Infatti ogni dispositivo è tenuto a comunicare la sua destinazione ad un nodo che si prende carico dell'elaborazione e successivamente informa i dispositivi del percorso con cui instradare i pacchetti. Qualora la struttura della rete fosse compatibile con la presenza di un nodo-elaboratore, questa soluzione centralizzata sarebbe da preferire a tutte le altre.

I metodi quasi distribuiti rappresentano un compromesso tra l'ottimalità garantita da un'elaborazione centralizzata e il ridotto *overhead* proprio delle soluzioni distribuite. Tra i metodi quasi-distribuiti il più completo è WHCA*: come abbiamo visto nella Sezione 4.4, questo algoritmo ci libera dalla dipendenza della soluzione dall'ordine con cui gli agenti eseguono il *planning*. Il principale vantaggio di questo approccio è quello di abbandonare un'elaborazione unificata dei cammini, e lasciare che ogni nodo pianifichi il percorso del suo pacchetto. Pur non garantendo di essere ottimi, con gli accorgimenti introdotti nel Capitolo 4, i metodi quasi distribuiti forniscono soluzioni complete e ottengono buone prestazioni (si veda [3] per i risultati sperimentali). Le problematiche sono le stesse dei metodi centralizzati: viene richiesta una comunicazione globale di controllo tra i dispositivi, ma l'informazione da condividere è sicuramente inferiore rispetto ai metodi centralizzati (vedi la Tabella 4.1 per una stima approssimativa).

Gli algoritmi distribuiti del Capitolo 5 sono adatti alle applicazioni in cui non è possibile uno scambio tra nodi che non siano adiacenti, cioè raggiungibili con una sola trasmissione. I cammini trovati sono sub-ottimi e spesso con costi più alti rispetto alle altre soluzioni; tuttavia questi algoritmi hanno il vantaggio di promuovere un comportamento cooperativo tra i nodi senza appesantire l'*overhead* della rete. Ogni nodo compie un *routing vorace* e gestisce le collisioni localmente con il metodo CSMA/CA della Sezione 5.1. L'algoritmo LRA* abbinato alla struttura DM, poi, introduce una maggiore distributività anche nella conoscenza della rete da parte dei dispositivi. Per la riuscita del *pathfinding*, infatti, ad ogni nodo è richiesto solo di mappare i dispositivi adiacenti e di stimare la distanza dagli altri nodi. Questa caratteristica può costituire un vantaggio per l'implementazioni di reti *ad hoc* molto dinamiche, in cui la topologia è soggetta a variazioni.

Bibliografia

[1] Matteo Fischetti. *Lezioni di Ricerca Operativa*. Libreria Progetto, 1999.

[2] M. Rossi C. Tapparello, S. Tomasin. On interference-aware cooperation policies for wireless ad hoc networks. *In attesa di pubblicazione*, 2010.

[3] David Silver. Cooperative pathfinding. *American Association for Artificial Intelligence*, 2005.

[4] Andrew S. Tanenbaum. *Computer Networks*. Pearson Education, 2003.

[5] Sven Koenig. Agent-centered search. *AI Magazine, Volume 22, Number 4*, 2001.

[6] N. Sturtevant R. Jansen. A new approach to cooperative pathfinding(short paper). *Proc. of 7th Int. Conf. on Autonomous Agents and Multiagent Systems*, 2008.

[7] Craig W. Reynolds. Flocks, herds, and schools: A distributed behavioral model. *Computer Graphics, Volume 21, Number 4*, 1987.

[8] V. Maniezzo M. Dorigo and A. Colorni. Ant system: Optimization by a colony of cooperating agents. *IEEE Transactions on Systems, man, and Cybernetics*, 1996.