Teoria dei grafi e la ricerca del cibo delle formiche

Menezio

15 aprile 2025

Indice

Introduzione

Viviamo in un mondo di connessioni. Ogni giorno, senza rendercene conto, interagiamo con strutture che possono essere modellate come **grafi**. Intuitivamente, un grafo è una rappresentazione di oggetti e delle loro relazioni: è una struttura matematica formata da **nodi** (o *vertici*) e **collegamenti** (o *archi*) tra di essi. È uno strumento semplice ma potentissimo, che permette di modellare reti, connessioni, movimenti, interazioni. Obiettivo di queste pagine, è quello di introdurre alla teoria dei grafi e ai suoi algoritmi, mostrando come questa disciplina permetta la modellizzazione e la "spiegazione" del comportamento di sistemi complessi. In particolar modo ci occuperemo del problema della ricerca di cibo delle formiche.

Nella sezione 1 iniziamo con esempi ludici che mostrano come i grafi possano essere utilizzati per modellare e risolvere problemi apparentemente complessi. A seguire, nella sezione 2 definiamo formalmente i grafi e introduciamo alcuni concetti fondamentali. Nelle sezioni 3 e 4, ci occuperemo di algoritmi per la ricerca di cammini in un grafo. Prima esploreremo un algoritmo classico per la ricerca del cammino minimo in un grafo pesato: l'algoritmo di Dijkstra. In seguito analizzeremo l'Ant Colony Optimization (ACO), un algoritmo ispirato al comportamento delle formiche per la ricerca di cibo. Infine, nella sezione 5, metteremo a confronto i due algoritmi, evidenziando le differenze e i vantaggi di ciascuno.

1 Giochi e grafi

1.1 Il gioco dei 4 cavalli

Immaginate di avere quattro cavalli e la seguente, insolita, scacchiera.

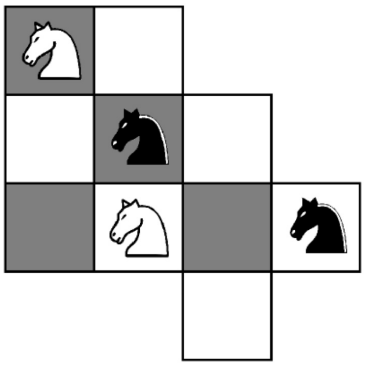

Figura 1: La scacchiera del gioco dei 4 cavalli.

Il problema che vogliamo porre è il seguente:

Data la scacchiera in Figura 1, è possibile scambiare di posizione i cavalli bianchi e quelli neri?

Le regole sono semplici: il cavallo si muove come nel gioco degli scacchi e può muoversi solo su una casella vuota.

Prima di andare avanti con la soluzione, è istruttivo provare a pensare da soli a una strategia. Ricordatevi che fare dei tentativi è a tutti gli effetti una strategia valida: spesso provando si scoprono regolarità inaspettate.

E se per caso riuscite nell'impresa, sapreste dire qual è il numero minimo di mosse necessarie a risolvere il rompicapo?

Spesso è utile capire per quale motivo il problema risulta difficile: in primis, il movimento degli scacchi a L può confonderci, inoltre, la ristrettezza della scacchiera non ci permette di muoverci

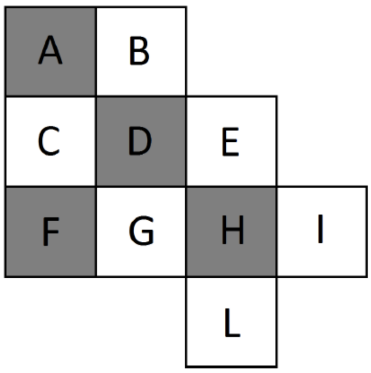

Figura 2: La scacchiera del gioco dei 4 cavalli con le caselle etichettate.

liberamente. Cerchiamo in un colpo solo di risolvere entrambi i problemi. Eticchiamo le caselle come in figura e proviamo a pensare a una strategia.

Costruiamo un grafo che catturi il movimento dei cavalli. I nodi del grafo sono le caselle della scacchiera e gli archi sono i movimenti possibili dei cavalli.

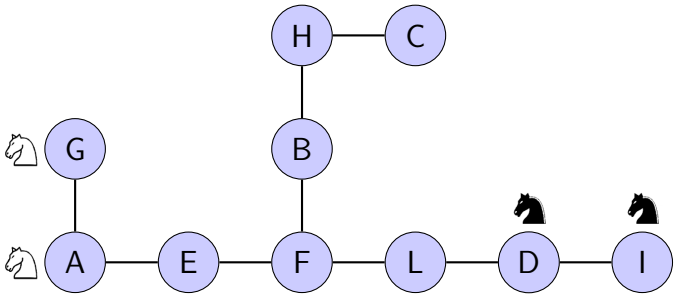

Figura 3: Il grafo che rappresenta i movimenti possibili dei cavalli sulla scacchiera.

La soluzione di questo problema è adesso immediata. I cavalli bianchi, ad esempio, si sposteranno nelle caselle B e H, permetteranno il passaggio dei cavalli neri e li sostituiranno nella loro posizioni. Anche la domanda per il numero minimo di mosse, che sembrava particolarmente complessa, è adesso facilmente approcciabile!

1.2 Il gioco degli smartphone

Il *gioco degli smartphone* è un interessante problema di disposizione in cui salta agli occhi la teoria dei grafi e la sua utilità.

Dati 4 smartphone, è possibile disporli in modo che ogni smartphone li tocca tutti tranne un altro?

Una possibile soluzione è quella di disporre i 4 smartphone in modo che formino i lati di un "quadrilatero" più grande.

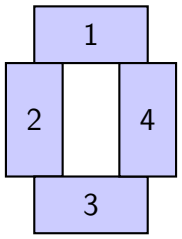

Figura 4: Una possibile soluzione del gioco dei 4 smartphone.

La difficoltà aumenta notevolmente se si aumenta il numero di smartphone. Ad esempio provate a risolvere lo stesso quesito con 5 smartphone:

Dati 5 smartphone, è possibile disporli in modo che ogni smartphone li tocca tutti tranne un altro?

Dopo un po' di tentativi, ci rendiamo conto che qualcosa non va e che sembra che non sia possibile. Dobbiamo però trovare un modo per formalizzare questa impossibilità. La prima semplificazione che possiamo fare è quella di immaginare gli smartphone come nodi e i contatti tra di loro come dei collegamenti tra nodi. Ecco che, astraendo il problema dalla realtà fisica, abbiamo una visualizzazione molto più chiara di quello che sta accadendo.

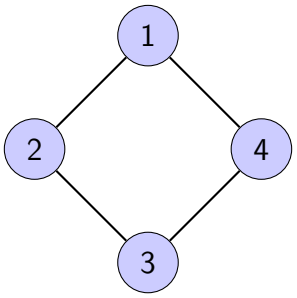

Figura 5: Il grafo associato alla soluzione del problema con 4 smartphone proposto sopra.

In questo modo, il problema si riduce a quello di trovare un grafo con 5 nodi in cui ogni nodo è collegato a esattamente 3 nodi.

Proviamo adesso a rispondere alla seguente domanda: quanti collegamenti devono essere presenti nel grafo per soddisfare la condizione del problema? Da ogni nodo partono 3 collegamenti, ma ogni collegamento collega due nodi. Questo vuol dire che il numero totale di collegamenti è uguale a 3 volte il numero di nodi diviso 2. In altre parole, il numero totale di collegamenti è uguale a $\frac{3\cdot 5}{2} = \frac{15}{2}$ che non è un numero intero! Dunque questo grafo non può esistere e il problema dei 5 smartphone è impossibile da realizzare!

Un semplice ragionamento di teoria dei grafi ci ha permesso di risolvere elegantemente il problema che adesso, grazie all'idea introdotta, si presta alla seguente generalizzaione:

Dati n smartphone con $n \geq 3$, è possibile disporli in modo che ogni smartphone li tocca tutti tranne un altro?

Lasciamo al lettore la soluzione del problema generale.

1.3 Non solo giochi

Nei due giochi precedenti abbiamo visto come l'utilizzo dei grafi permette di modellizzare due giochi apparentemente complicati che si rivelano essere molto più comprensibili con il giusto formalismo.

Al di là dei giochi, i grafi sono uno strumento potentissimo per la modellizzazione di problemi complessi; nel seguito di queste note ci occuperemo di alcuni esempi di applicazione della teoria dei grafi a problemi reali. In particolare, dopo aver formalizzato il concetto di grafo, ci soffermeremo su due applicazioni e sulle implicazioni pratiche che ne derivano:

- La ricerca del cammino minimo in un grafo pesato;
- La ricerca di cibo delle formiche.

2 Elementi di teoria dei grafi

In questa sezione, ci occupiamo di dare le definizioni fondamentali per la teoria dei grafi.

Definizione 1 *Un* grafo *è una coppia* $G = (V, E)$*, dove:*

- V *è un insieme finito di elementi chiamati* vertici *o* nodi*;*
- $E \subseteq \{\{u, v\} \mid u, v \in V,\ u \neq v\}$ *è un insieme di* archi*, ciascuno dei quali è una coppia non ordinata di vertici.*

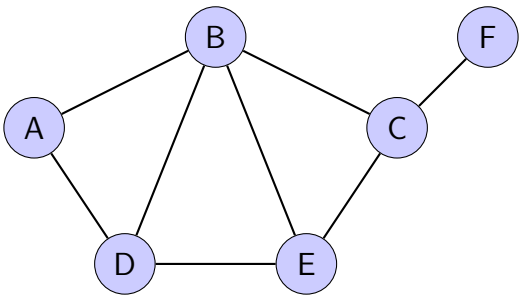

Figura 6: Esempio di grafo.

Osservazione 1 *Nella definizione di grafo che abbiamo dato non ammettimo l'esistenza di loop, ovvero di nodi che hanno un arco che parte e arriva allo stesso nodo. Inoltre, non ammettiamo archi multipli, ovvero più archi che collegano la stessa coppia di nodi. In altre parole, ogni coppia di nodi può essere connessa da al massimo un arco.*

Definizione 2 *Un* grafo orientato *è una coppia* $G = (V, A)$*, dove:*

- V *è un insieme finito di vertici;*
- $A \subseteq \{(u, v) \mid u, v \in V,\ u \neq v\}$ *è un insieme di* archi orientati*, cioè coppie ordinate di vertici.*

Il fatto che i vertici siano coppie ordinate permette un verso di percorrenza degli archi. In altre parole, l'arco (u, v) è l'arco che parte dal nodo u e arriva al nodo v ed è l'arco *opposto* a (v, u).

Se non specificato, un grafo verrà consideratto non orientato.

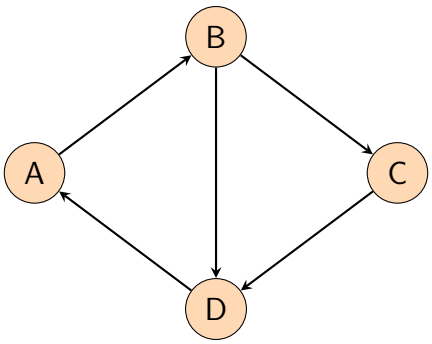

Figura 7: Esempio di grafo orientato.

Definizione 3 *Un* grafo pesato *è un grafo (orientato o non orientato) in cui a ciascun arco è associato un valore numerico, detto* peso *o* costo*. Formalmente, un grafo ponderato è una terna* $G = (V, E, w)$*, dove:*

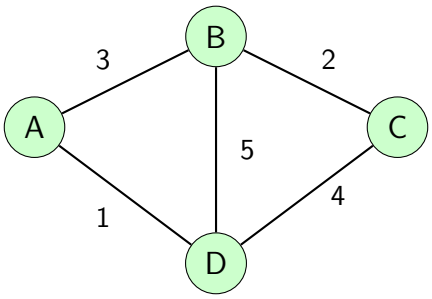

Figura 8: Esempio di grafo pesato con pesi sugli archi.

- (V, E) *è un grafo;*
- $w : E \to \mathbb{R}$ *è una funzione che assegna un peso reale a ciascun arco.*

Definizione 4 *Un* cammino *in un grafo* $G = (V, E)$ *è una sequenza di vertici* $(v_0, v_1, \ldots, v_k)$ *tale che per ogni* $i = 0, \ldots, k-1$*, l'arco* $\{v_i, v_{i+1}\} \in E$*.*

In altre parole, un cammino non è altro che una sequenza di vertici in cui ogni coppia consecutiva di vertici è connessa da un arco.

Definizione 5 *Un* cammino semplice *o* cammino senza cicli *è un cammino in cui tutti i vertici sono distinti, cioè* $v_i \neq v_j$ *per ogni* $i \neq j$*.*

Definizione 6 *Un* ciclo *è un cammino* $(v_0, v_1, \ldots, v_k)$ *con* $k \geq 2$*, tale che:*

- $v_0 = v_k$ *(il cammino inizia e termina nello stesso vertice);*
- *i vertici* $v_0, \ldots, v_{k-1}$ *sono distinti.*

Un ciclo è quindi un cammino che inizia e termina nello stesso vertice, ma non attraversa lo stesso vertice più di una volta (eccetto il primo e l'ultimo).

Definizione 7 *Un grafo non orientato è detto* connesso *se esiste un cammino tra ogni coppia di vertici.*

Definizione 8 *Un* albero *è un grafo connesso e aciclico, cioè che non contiene cicli.*

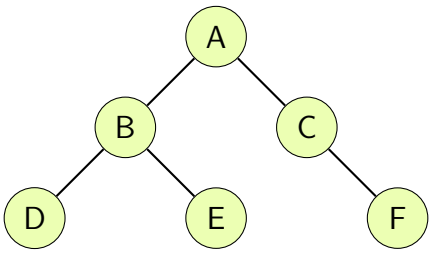

Figura 9: Un albero: grafo connesso senza cicli.

Nella prossima sezione, cominceremo a parlare di algoritmi sui grafi per l'esplorazione e per la ricerca di cammini. Siamo in particolare interessati ai cammini minimi.

Definizione 9 *Dato un grafo pesato* $G = (V, E, w)$*, un* cammino minimo *tra due vertici è un cammino tale che la somma dei pesi degli archi attraversati è minima.*

3 Ricerca di cammini minimi sui grafi; l'algoritmo di Dijkstra

Una domanda che sorge spontanea quando ci troviamo di fronte a un grafo pesato è quale sia il cammino più breve tra due nodi. In altre parole, vogliamo trovare il cammino che minimizza la somma dei pesi degli archi attraversati.

L'algoritmo di Dijkstra è uno dei più noti algoritmi per la ricerca del *cammino minimo* in un grafo pesato. È stato proposto da Edsger W. Dijkstra nel 1956 e fornisce un metodo efficiente per calcolare la distanza minima tra un nodo sorgente e tutti gli altri nodi di un grafo con pesi non negativi.

Definizione 10 *Sia $G = (V, E)$ un grafo orientato e pesato, dove V è l'insieme dei nodi e E è l'insieme degli archi con una funzione peso $w : E \to \mathbb{R}_{\geq 0}$ che assegna un costo non negativo ad ogni arco. Sia $s \in V$ il nodo sorgente. L'algoritmo di Dijkstra calcola per ogni nodo $v \in V$ la distanza dist$[v]$ minima da s a v, ovvero il peso del cammino più breve.*

L'algoritmo di Dijkstra costruisce passo dopo passo il risultato corretto, seguendo una logica **greedy**: ad ogni passo seleziona la scelta più promettente nel breve termine, con la garanzia di ottenere comunque la soluzione ottima globale.

3.1 Struttura e funzionamento

Il cuore dell'algoritmo è una struttura dati che tiene traccia delle **distanze minime note** tra il nodo sorgente e ogni altro nodo del grafo. All'inizio, questa struttura contiene valori pari a ∞ per ogni nodo tranne il nodo sorgente s, che ha distanza 0. Si mantiene anche un insieme (o una coda di priorità) dei nodi ancora da esplorare.

Il funzionamento può essere descritto così:

1. Si inizia dal nodo sorgente s, e si considera la sua distanza come definitiva (cioè: si "visita" il nodo).

2. Si esaminano tutti i nodi adiacenti a s (i suoi *vicini*) e si aggiorna la loro distanza provvisoria: per ogni vicino v, si calcola il valore dist$[s] + w(s, v)$ e si confronta con dist$[v]$. Se è più piccolo, si aggiorna.

3. Tra tutti i nodi non ancora visitati, si sceglie quello con la distanza provvisoria più bassa, diciamo u, e lo si marca come **definitivamente visitato**. Questo significa che non esiste alcun altro cammino più corto da s a u: la distanza è quella minima garantita.

4. Si ripete il processo: si aggiornano i vicini di u, e si passa poi al nodo con la distanza provvisoria minima tra quelli ancora non visitati.

5. Si procede fino a che tutti i nodi sono stati visitati, o fino a che la distanza provvisoria minima tra i rimanenti nodi è ∞ (cioè non raggiungibili da s).

Perché l'algoritmo garantisce il cammino minimo?

La correttezza dell'algoritmo si basa su due osservazioni fondamentali:

- **Invarianza del minimo:** ogni volta che un nodo u viene "visitato" (cioè, la sua distanza viene marcata come definitiva), possiamo essere certi che la distanza trovata è **la minima possibile**. Questo perché, se esistesse un altro cammino più corto verso u, quel cammino dovrebbe passare da nodi che ancora non sono stati visitati. Ma dato che scegliamo sempre il nodo con distanza provvisoria minima, tale cammino non potrebbe avere costo inferiore a quello già trovato.

- **Assenza di pesi negativi:** se esistessero archi con peso negativo, un nodo già visitato potrebbe avere un cammino alternativo più corto scoperto successivamente, invalidando l'invarianza del minimo. Per questo motivo, Dijkstra funziona solo se tutti i pesi degli archi sono non negativi.

In sostanza, Dijkstra costruisce una "sfera" di nodi visitati intorno al nodo sorgente, espandendo iterativamente l'area raggiunta e aggiornando le distanze. Una volta che un nodo entra in questa sfera, la sua distanza è fissata per sempre.

Osservazione 2 *Si può immaginare l'algoritmo come un'onda che si espande dal nodo sorgente, raggiungendo prima i nodi più vicini (in termini di costo) e poi quelli più lontani. L'espansione è ordinata in base alla distanza minima, e questo garantisce che ogni nodo venga raggiunto dal cammino più economico.*

Esempio illustrato

Vediamo un esempio pratico dell'algoritmo di Dijkstra applicato a un grafo pesato. Consideriamo il grafo seguente, in cui i numeri sugli archi rappresentano i pesi:

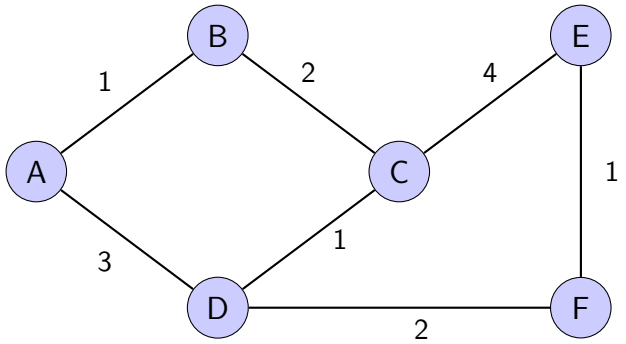

Figura 10: Grafo di esempio per Dijkstra.

Nel grafo dell'esempio:

- Inizializziamo le distanze: dist$[A] = 0$, dist$[B] = \infty$, dist$[C] = \infty$, dist$[D] = \infty$, dist$[E] = \infty$, dist$[F] = \infty$.
- Partiamo da A con dist$[A] = 0$.
- Da A, possiamo andare a B con costo 1 o a D con costo 3. Quindi dist$[B] = 1$, dist$[D] = 3$.
- Il nodo con distanza minore è B, lo visitiamo: da B si può andare a C con costo $1 + 2 = 3$, quindi dist$[C] = 3$.
- I nodi con distanza minore sono C e D. Ne scegliamo arbitrariamente uno e lo visitiamo (questa scelta non influisce sulla correttezza del metodo come osservato sopra). In questo esempio scegliamo D: da D si può andare a F con costo $3+2 = 5$, quindi dist$[F] = 5$. Inoltre, da D si può andare a C con costo $3 + 1 = 4$, ma dist$[C]$ è già 3, quindi non aggiorniamo.
- Il nodo con distanza minore adesso è C, lo visitiamo: da C si può andare a E con costo $3 + 4 = 7$, quindi dist$[E] = 7$.
- Il nodo con distanza minore è F, lo visitiamo: da F si può andare a E con costo $5 + 1 = 6$, quindi dist$[E] = 6$.
- Infine, visitiamo E, ma non ci sono aggiornamenti da fare.
- Distanze finali: dist$[A] = 0$, dist$[B] = 1$, dist$[C] = 3$, dist$[D] = 3$, dist$[E] = 6$, dist$[F] = 5$.

3.2 Implementazione e sperimentazione

Abbiamo implementato l'algoritmo di Dijkstra in Python, utilizzando una coda di priorità per ottimizzare la selezione del nodo con distanza minima. L'implementazione è stata testata su grafi di piccole e medie dimensioni, e riportiamo di seguito i link ad alcune visualizzazioni che mostrano il funzionamento dell'algoritmo in tempo reale.

!!! Inserisci i link !!!

Qualche considerazione

L'algoritmo di Dijkstra è una soluzione ottimale per grafi con pesi non negativi. Tuttavia, non è adatto per grafi con archi a peso negativo: in tali casi, è preferibile usare l'algoritmo di Bellman-Ford[1]. Inoltre, Dijkstra è un algoritmo deterministico e converge sempre al cammino più breve. Questo è ottimo per situazioni in cui la certezza è fondamentale, ma può essere limitante in scenari in cui si desidera esplorare soluzioni alternative o si ha a che fare con dati incerti o che magari cambiano nel tempo.

4 Dalle formiche ai grafi: ottimizzazione tramite intelligenza collettiva

4.1 Il problema della ricerca del cibo

Nel mondo naturale, uno dei comportamenti più affascinanti è quello delle formiche alla ricerca del cibo. Nonostante siano organismi molto semplici, prive di una vera intelligenza individuale, sono in grado di risolvere problemi complessi grazie alla cooperazione e alla comunicazione indiretta.

In particolare, le formiche riescono a trovare percorsi ottimali tra il nido e una fonte di cibo. Non possiedono mappe né un senso diretto delle distanze, ma riescono a scoprire e sfruttare cammini efficienti attraverso un meccanismo di **comunicazione chimica**, basato sui feromoni.

Questo comportamento ha ispirato un intero filone dell'intelligenza artificiale chiamato *Ant Colony Optimization* (ACO), in cui si cerca di riprodurre artificialmente il comportamento delle colonie di formiche per risolvere problemi di ottimizzazione su grafi.

Formalizzazione del problema

Il comportamento delle formiche può essere modellato attraverso un grafo, in cui:

- I **nodi** rappresentano posizioni fisiche (come il nido, i punti di passaggio e le fonti di cibo).
- Gli **archi** rappresentano i percorsi possibili tra le posizioni.
- Ogni arco ha un **peso**, che può rappresentare una distanza, un costo, o una difficoltà nel percorrerlo.

Il problema che ci poniamo è trovare **cammini convenienti** tra due nodi del grafo: il nodo sorgente (il nido) e il nodo obiettivo (una tra le fonti di cibo).

4.2 Algoritmo delle colonie di formiche - *Ant Colony Optimization* o ACO

Principi generali

Il modello ACO si basa su alcuni principi fondamentali, ispirati al comportamento reale:

1. **Deposizione di feromoni:** ogni formica, mentre percorre un cammino, deposita una quantità di feromone sugli archi attraversati.

[1] Aggiungi referenza

2. **Evaporazione:** i feromoni evaporano col tempo, riducendo la loro intensità.

3. **Scelta probabilistica:** una formica decide quale nodo visitare in base a una probabilità che dipende:

   - dalla quantità di feromone presente sugli archi;
   - da un'informazione euristica (ad esempio, l'inverso della distanza). Prenderemo proprio questo esempio nel prosieguo[2].

4. **Rinforzo positivo:** i cammini migliori, ricevono più feromoni in tempi brevi, e diventano quindi più attraenti per altre formiche.

Formalizzazione matematica

Sia $\tau_{ij}(t)$ la quantità di feromone sull'arco (i, j) al tempo t (più precisamente all'iterazione t dell'algoritmo), e η_{ij} l'informazione euristica associata (ad esempio $\eta_{ij} = \frac{1}{d_{ij}}$, dove d_{ij} è la lunghezza dell'arco).

La probabilità che una formica attualmente in i scelga di muoversi verso il nodo j è data da:

$$P_{ij}(t) = \frac{[\tau_{ij}(t)]^{\alpha} \cdot [\eta_{ij}]^{\beta}}{\sum_{k \in N_i} [\tau_{ik}(t)]^{\alpha} \cdot [\eta_{ik}]^{\beta}}$$

dove:

- N_i è l'insieme dei nodi vicini a i ancora visitabili;
- α regola l'influenza del feromone;
- β regola l'influenza dell'euristica.

Osservazione 3 *La probabilità è una media pesata tra il feromone e l'euristica. Il parametro α controlla quanto le formiche si affidano al feromone, mentre β regola l'importanza dell'euristica. Questo bilanciamento permette di esplorare nuovi cammini senza trascurare quelli già promettenti.*

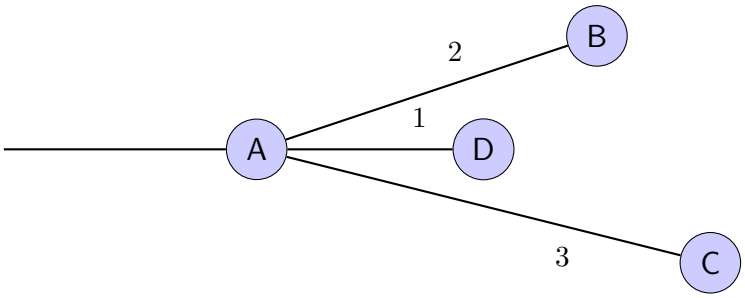

Figura 11: Un grafo con nodo A e tre archi in uscita. Supponendo feromoni $\tau_{AB} = 3$, $\tau_{AC} = 2$, $\tau_{AD} = 4$ e usando l'inverso della distanza come euristica ($\eta_{ij} = 1/w$), calcoliamo le probabilità. I parametri utilizzati sono $\alpha = 1$ e $\beta = 1$. Il denominatore è dato da $\sum_{k \in N_A} [\tau_{Ak}]^{\alpha} \cdot [\eta_{Ak}]^{\beta} = (3 \cdot 1/2) + (2 \cdot 1/3) + (4 \cdot 1/1) = 1.5 + 0.6667 + 4 = 6.1667$. Le probabilità risultano: $P_{AB} \approx 0.24$, $P_{AC} \approx 0.11$, $P_{AD} \approx 0.65$.

Aggiornamento dei feromoni:

Dopo che tutte le formiche hanno completato il loro cammino, la quantità di feromone su ogni arco viene aggiornata secondo:

[2] Si possono aggiungere parametri di pericolosità o affidabilità del cammino. L'implementazione è analoga a quanto viene fatto con la distanza

$$\tau_{ij}(t+1) = (1-\rho) \cdot \tau_{ij}(t) + \sum_{k=1}^{m} \Delta\tau_{ij}^{(k)}$$

dove:

- ρ è il tasso di evaporazione $(0 < \rho < 1)$,
- $\Delta\tau_{ij}^{(k)}$ è la quantità di feromone depositata dalla k-esima formica.

La quantità depositata dalla k-esima formica è proporzionale alla qualità del percorso:

$$\Delta\tau_{ij}^{(k)} = \begin{cases} \frac{Q}{L_k} & \text{se l'arco } (i,j) \text{ è stato percorso dalla formica } k \\ 0 & \text{altrimenti} \end{cases}$$

dove L_k è la lunghezza totale del percorso della formica k, e Q è una costante positiva.

4.3 Implementazione e sperimentazione

Abbiamo sperimentato il funzionamento dell'algoritmo ACO su vari grafi pesati, utilizzando Python e librerie come NetworkX per la gestione dei grafi e Matplotlib per la visualizzazione. L'implementazione è stata testata su grafi di piccole e medie dimensioni; alla fine di questa sezione riportiamo i link a delle visualizzazioni che mostrano il funzionamento dell'algoritmo in tempo reale.

Di seguito facciamo una panoramica dei dati che vengono raccolti ad ogni esecuzione dell'algoritmo. Le immagini riportate mostrano il grafo iniziale, il grafo con evidenziata la distribuzione del feromone sugli archi a una iterazione generica, la distribuzione percentuale dei feromoni nei vari archi a una iterazione generica e l'andamento del cammino di una formica. Le formiche partono dal nodo sorgente (evidenziato in rosso) e si muovono verso le fonti di cibo (evidenziate in verde), depositando feromone lungo il percorso. Durante le iterazioni, alcuni nodi vengono eliminati ed altri si riaggiungono: l'algoritmo si adatta di conseguenza restituendo sempre ottimi risultati anche in situazioni di grafo dinamico.

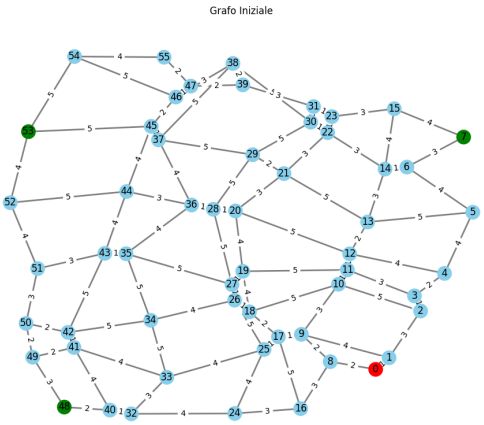

Figura 12: Grafo iniziale con pesi sugli archi. Il nodo rosso rappresenta il nido, i nodi verdi le fonti di cibo

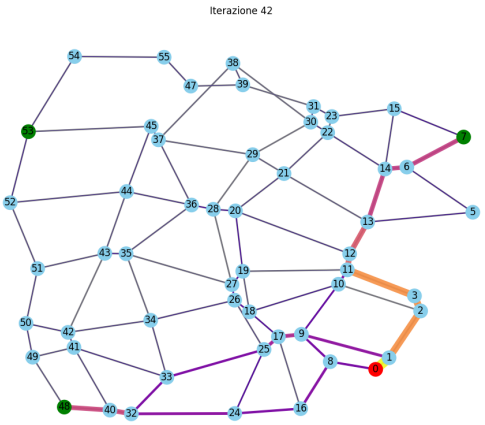

Figura 13: Distribuzione del feromone sugli archi all'iterazione 42. I colori più spessi e accesi indicano la presenza di feromone negli archi.

Più nel dettaglio, l'algoritmo ACO è stato implementato con vari parametri per poter testare la sua efficacia in diverse situazioni. Di seguito, riportiamo una breve descrizione dei parametri più importanti. A questi, vanno aggiunti creazioni di grafi casuali e configurazioni che regolano la dinamicità del grafico.

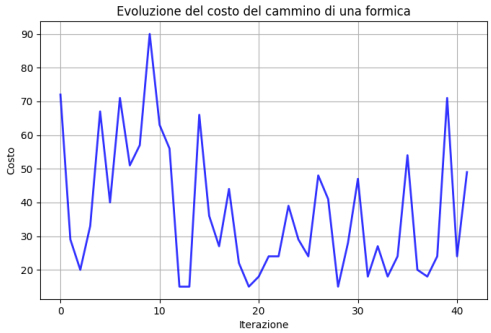

Figura 14: Andamento del percorso di una formica nel corso delle iterazioni del ciclo.

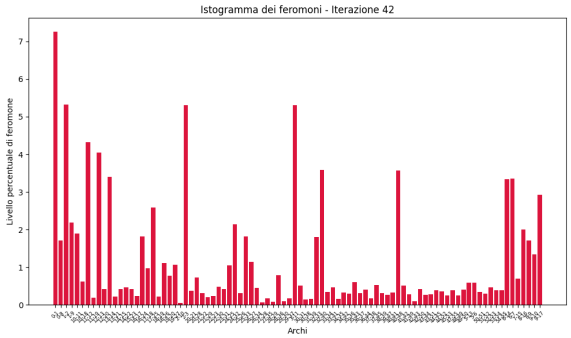

Figura 15: Distribuzione percentuale dei feromoni nei vari archi.

- `NUM-FORMICHE`: Numero di formiche utilizzate nell'algoritmo. Ad ogni iterazione, partono dal nido un numero di formiche pari al valore indicato. Valori alti aumentano l'esplorazione in quanto la scelta dei percorsi, anche se vincolata alla probabilità, risulta più ampia. Questo va a scapito dell'esecuzione; valori bassi, invece, riducono la diversità delle soluzioni.
- `NUM-ITERAZIONI`: Numero totale di iterazioni dell'algoritmo. Valori alti migliorano la qualità della soluzione ma aumentano il tempo di calcolo; valori bassi possono portare a soluzioni subottimali. Nelle applicazioni reali, le iterazioni vengono lanciate senza un valore massimo fissato, In questo modo, se il grafo cambia dinamicamente, i cammini migliori vengono ricalcolati in tempo reale.
- `ALFA` α: Peso attribuito al feromone nella scelta del cammino. Valori alti favoriscono lo sfruttamento dei cammini già esplorati, rendendo il comportamento delle formiche più conservativo e focalizzato sui percorsi che hanno già dimostrato di essere buoni. Questo può accelerare la convergenza verso una soluzione, ma rischia di bloccare il sistema in minimi locali. Valori bassi, invece, aumentano l'esplorazione di nuovi cammini, favorendo una maggiore diversità nelle soluzioni, ma rallentando la convergenza.
- `BETA` β: Peso attribuito all'euristica (es. distanza) nella scelta del cammino. Valori alti danno maggiore importanza all'euristica, spingendo le formiche a scegliere cammini che sembrano promettenti in base a informazioni statiche come la distanza o il costo. Questo può essere utile in grafi con pesi ben definiti e affidabili. Valori bassi bilanciano l'influenza tra feromone ed euristica, permettendo alle formiche di considerare sia l'esperienza accumulata (feromone) sia le caratteristiche intrinseche del grafo (euristica).
- `EVAPORAZIONE` ρ: Tasso di evaporazione del feromone (valore tra 0 e 1). Questo parametro controlla la velocità con cui il feromone depositato sugli archi diminuisce nel tempo. Valori alti (vicini a 1) fanno sì che le tracce di feromone scompaiano rapidamente, favorendo l'esplorazione di nuovi cammini e riducendo l'influenza delle soluzioni precedenti. Valori bassi (vicini a 0) mantengono più a lungo le tracce di feromone, rafforzando i cammini già esplorati e favorendo lo sfruttamento delle soluzioni esistenti. La scelta del valore dipende dal bilanciamento desiderato tra esplorazione e sfruttamento.
- `FEROMONE-INIZIALE`: Quantità iniziale di feromone su ogni arco. Valori alti favoriscono una scelta più uniforme inizialmente; valori bassi aumentano la casualità.
- `FEROMONE-DEPOSITO` Q: Quantità di feromone depositata dalle formiche. Valori alti rafforzano rapidamente i cammini migliori; valori bassi favoriscono una convergenza più lenta.
- `NODO-PARTENZA`, `NODI-DESTINAZIONE`: Nodi che identificano il nido e le fonti di cibo

4.4 Applicazioni dell'algoritmo ACO

L'algoritmo ACO è stato applicato con successo a numerosi problemi di ottimizzazione combinatoria, tra cui:

- il **problema del commesso viaggiatore** (TSP): trovare il percorso più breve che permette a un venditore di visitare un insieme di città una sola volta e tornare al punto di partenza. Questo problema è fondamentale in logistica e trasporti, dove l'ottimizzazione dei percorsi può ridurre significativamente i costi operativi e i tempi di consegna;
- il **routing in reti** di telecomunicazione: ottimizzare il flusso di dati attraverso una rete per minimizzare ritardi e congestioni. Questo è cruciale per garantire la qualità del servizio in applicazioni come streaming video, chiamate VoIP e trasferimenti di file su larga scala;
- la **pianificazione** di operazioni industriali: organizzare sequenze di operazioni per migliorare l'efficienza produttiva e ridurre i tempi morti. Ad esempio, nelle catene di montaggio, una pianificazione ottimale può aumentare la produttività e ridurre i costi di manutenzione;
- il **pathfinding** in robotica e videogiochi: determinare il percorso ottimale per un robot o un personaggio virtuale per raggiungere un obiettivo evitando ostacoli. Questo è essenziale per applicazioni come la navigazione autonoma di robot in ambienti complessi o l'intelligenza artificiale nei giochi per offrire esperienze realistiche e coinvolgenti.

Il grande pregio dell'ACO è la sua capacità di trovare buone soluzioni anche in contesti in cui non esistono strategie deterministiche efficienti, sfruttando la ridondanza, la parallelizzazione e la selezione delle soluzioni migliori.

5 ACO vs Dijkstra: Obiettivi e Differenze

In questa ultima sezione, vogliamo mettere a confronto i due algoritmi presentati: l'algoritmo di Dijkstra e l'algoritmo ACO. Entrambi sono progettati per affrontare problemi di ottimizzazione su grafi, ma si differenziano notevolmente nei loro approcci e nelle loro applicazioni.

Nella seguente tabella sono riportati alcuni aspetti chiave che differenziano i due algoritmi:

Caratteristica	**Dijkstra**	**ACO**
Tipo di algoritmo	Deterministico, esatto	Probabilistico, euristico
Obiettivo	Cammino minimo unico	Buoni cammini multipli
Complessità computazionale	$\mathcal{O}(n \log n)$	Dipende dalle iterazioni
Conoscenza del grafo	Completa	Locale (feromone, euristica)
Adattabilità	Bassa	Alta, dinamica
Ispirazione	Matematica classica	Natura (formiche)
Scenari ideali	Reti statiche, costi noti	Ambienti dinamici, complessi

Tabella 1: Confronto tra Dijkstra e ACO (Ant Colony Optimization).

5.1 L'obiettivo dell'ACO non è la convergenza rigida

A differenza di molti algoritmi classici, il comportamento delle formiche non punta a convergere rapidamente a una soluzione unica e definitiva. L'obiettivo dell'ACO è piuttosto mantenere un bilanciamento tra:

- **Esplorazione** di nuovi cammini potenzialmente migliori;
- **Sfruttamento** dei percorsi che si sono rivelati buoni.

Questa tensione tra curiosità e fiducia è controllata dai parametri α, β, e ρ (rispettivamente peso del feromone, dell'euristica e della velocità di evaporazione).

In questo senso, una convergenza rapida e rigida su un cammino specifico può essere controproducente. Infatti, se le formiche si concentrano troppo presto su un solo percorso, il sistema rischia di bloccarsi in un minimo locale, perdendo la possibilità di esplorare soluzioni migliori.

5.2 Quando scegliere ACO rispetto a Dijkstra

In generale, l'algoritmo delle formiche è preferibile quando:

- il grafo cambia nel tempo o contiene incertezze (es. traffico, costi variabili);
- si cercano molte buone soluzioni, non una sola perfetta;
- si vuole una soluzione distribuita e scalabile;
- si affrontano problemi combinatori complessi (es. TSP, vehicle routing, scheduling).

In ambienti controllati e statici, Dijkstra rimane insuperabile in termini di efficienza e precisione. Ma in molti contesti reali, la natura dinamica e flessibile dell'ACO lo rende una scelta più realistica e potente.