Alma Mater Studiorum · Università di Bologna

SCUOLA DI SCIENZE

Corso di Laurea in Matematica

Teoria dei grafi applicata allo studio dei labirinti

Relatore:
Dott.ssa
Alessia Cattabriga

Presentata da:
Tommaso Saracchini

II Sessione
Anno Accademico 2016/2017

Introduzione

Il labirinto ha accompagnato la storia dell'uomo tra culture, epoche e luoghi; il simbolo del labirinto compare già in epoca preistorica e nel corso del tempo la sua struttura ed il suo significato sono cambiate.
Ai tempi del medioevo il labirinto aveva un significato "spirituale", ad esempio, per alcuni, la strada al suo interno simboleggiava il percorso della vita, quindi nascita, crescita e infine morte o, per altri, stava a rappresentare il cammino di espiazione dei peccati per coloro che non potevano recarsi in Terra Santa, ragion per cui veniva spesso raffigurato sul pavimento all'interno delle cattedrali.
A partire dall'età rinascimentale il labirinto perde questa connotazione spirituale, esso si trasforma in un semplice rompicapo, tanto ingannevole quanto intrigante; allo stesso tempo cambia anche la sua struttura, che da unicursale, cioè caratterizzata da un percorso spiraliforme che si avvolge fino al centro, diventa multicursale, cioè contiene più strade che conducono alla medesima uscita (per approfondimenti si veda [3]).

Mentre l'origine dei labirinti si perde nella preistoria, la teoria dei grafi ha una data di nascita precisa: il primo lavoro sui grafi fu scritto dal matematico svizzero Leonhard Euler, ed apparve nel 1736. Il lavoro di Eulero conteneva la soluzione del cosiddetto *problema dei ponti di Königsberg*, in cui ci si chiede se sia possibile con una passeggiata seguire un percorso che attraversasse ogni ponte della città una volta soltanto.
I grafi concettualmente esprimono relazioni binarie tra oggetti e modelizzano

perfettamente la struttura interna del labirinto, vale a dire che lo descrivono in termini di "percorribilità", dando quindi informazioni sulle strade e gli incroci presenti nel labirinto e contrubuendo a formalizzare matematicamente problemi come, ad esempio, costruire un labirinto o uscire da un labirinto.
Nel primo capitolo di questa tesi definiremo informalmente i labirinti e li suddivideremo in diverse classi in base a determinate caratteristiche.
Nel secondo capitolo esporremo la teoria dei grafi, in particolare tutto ciò che sarà utile per descrivere un labirinto, tra cui la definizione di cammino, connessione, isomorfismo, il problema di planarità e gli algoritmi di visita di un grafo.
Nel terzo capitolo andremo ad utilizzare la teoria dei grafi per descrivere, generare o risolvere un labirinto generico.
Infine, nel quarto ed ultimo capitolo, andremo ad estendere il concetto di labirinto, quindi vedremo come applicare la teoria vista nei capitoli precedenti a problemi di diversa natura come reti informatiche o stradali, ma tutti accomunati dal fatto di poter essere ben modellizzati da un grafo.

Indice

Elenco delle figure

Capitolo 1

Il labirinto

In questo capitolo definiremo informalmente i labirinti e li suddivideremo in diverse classi in base a determinate caratteristiche (per approfondimenti si veda [4]).

1.1 Cos'è un labirinto

Il *labirinto*, il gioco in cui s'imbattono spesso gli appassionati di enigmistica o di videogiochi, è in realtà un emblema antichissimo, universalmente conosciuto, giunto a noi attraverso millenni di storia, che compare in tempi e luoghi molto distanti tra di loro.
Andiamo ad esporre dei termini di cui faremo uso per descrivere un labirinto, si veda Figura 1.1. Esso è composto da *celle* e *muri*, celle adiacenti possono essere collegate tra loro oppure separate da un muro. Tra le celle, due di esse saranno denominate una *cella di entrata* e l'altra *cella di uscita* (esse possono anche coincidere); perciò il problema collegato ad un labirinto è quello di trovare un percorso tramite celle collegate che, partendo da una data cella, mi permetta di arrivare alla cella di uscita (nel caso coincidano generalmente lo scopo è quello di, una volta entrati, raggiungere una data cella "obiettivo" per poi ritornare alla cella di partenza). Quando una cella è collegata a più di due celle viene detta *incrocio* (in giallo in figura), mentre se

è collegata solamente ad un altra, allora il percorso che porta da quest'ultima cella all'incrocio più vicino è detto *vicolo cieco* (in rosso in figura). Infine se partendo da una cella esiste un percorso, composto da celle differenti, che mi permetta di ritornare a quella di partenza, esso viene chiamato *ciclo*.

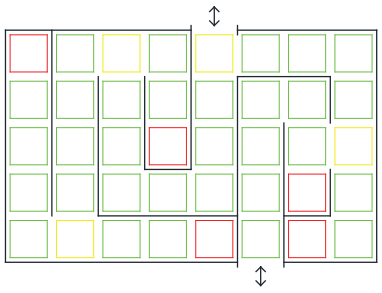


Figura 1.1: Un labirinto con muri in nero e celle quadrate: le celle gialle rappresentano gli incroci, quelle rosse i vicoli ciechi e le verdi le rimanenti. Le frecce indicano entrata e/o uscita.

La struttura originaria del labirinto è unicursale, cioè costituita da un percorso spiraliforme che si avvolge fino a raggiungere il centro; in questi labirinti non è difficile arrivare al centro poiché non ci sono incroci e anche l'uscita è altrettanto facile da trovare percorrendo il percorso in senso inverso.
Attualmente invece, il labirinto viene considerato come un rompicampo, quindi con una struttura multicursale dove, a differenza dei labirinti antichi, sono presenti incroci. Proprio per questo fatto, una volta entrati, ci si trova a compiere una serie di decisioni su quale percorso intraprendere per raggiungere l'uscita.
In questa tesi andremo a studiare solo questi ultimi dato che, in termini di teoria dei grafi, gli unicursali sono poco interessanti. Di seguito per labirinto intendiamo dunque un labirinto multicursale.

1.2 Tipologie di labirinto

I labirinti possono essere classificati in base a diversi criteri, di seguito ne riportiamo alcuni:

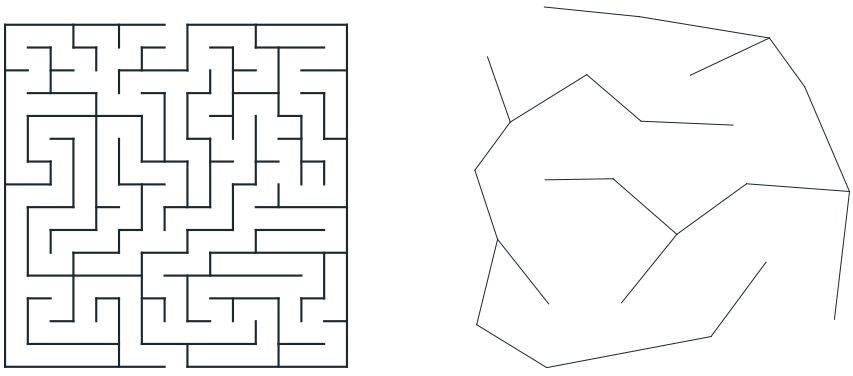
dimensione, **tassellazione**, **percorso** e **topologia**.

Dimensione: questa classe ci dice su quante dimensioni si sviluppa il labirinto.

- *2D*: sono i labirinti sviluppabili in una superficie bidimesionale, caratterizzati dal fatto di avere poligoni come celle. La maggior parte dei labirinti rientra in questa fascia e per questo ci concentreremo maggiormente su di essi;

- *3D o più*: sono i labirinti che si sviluppano in 3 o più dimensioni, essi necessitano di poliedri o politopi n-dimensionali come celle per essere descritti.

Tassellazione: la classe di tassellazione dipende dalla forma delle singole celle che compongono il labirinto, vedremo che un modo per costruire labirinti è infatti usare una tassellazione vedi Sezione 3.2.
Spesso considereremo tassellazioni *regolari*, ovvero realizzate con poligoni regolari come celle, come ad esempio il labirinto in Figura 1.2(a) che ha celle quadrate, tuttavia possiamo estendere il concetto a tassellazioni qualunque, ad esempio le celle del labirinto in Figura 1.2(b) sono poligoni generici.

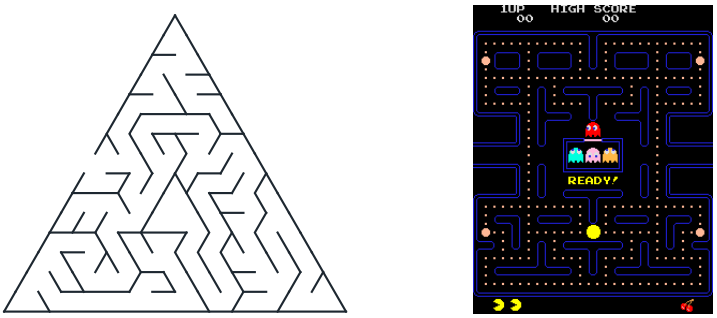


(a) *Regolare* (b) *Irregolare*

Figura 1.2: Esempi di labirinti con diversa tassellazione.

Percorso: questo tipo di classificazione fa riferimento al tipo di percorsi presenti all'interno di un labirinto, vedi Figura 1.3.

- *Perfetto*: un labirinto è perfetto quando non contiene cicli. Esso viene anche chiamato "semplicemente connesso". Un fatto importante di questi labirinti è che al loro interno sono presenti solamente vicoli ciechi e questo comporta l'esistenza di un unico percorso per uscirne;
- *Braid*: si chiama braid un labirinto dove non ci sono vicoli ciechi, quindi al contrario dei "perfetti" esso contiene cicli (un braid ben fatto è molto piu difficile da risolvere di uno perfetto).



(a) *Perfetto* (b) *Braid*

Figura 1.3: Esempi di percorsi nel labirinto.

Topologia: questa classificazione fa riferimento al tipo di superficie o spazio su cui è disegnato il labirinto, vedi Figura 1.4.
In particolare, ci interesseremo di labirinti sviluppati su superfici sia piane sia curve (come la sfera o il toro).

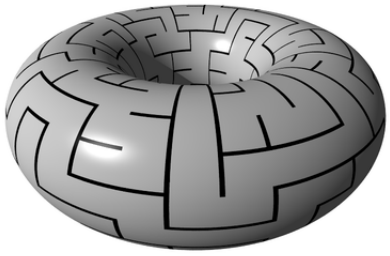


Figura 1.4: Esempio di labirinto toroidale.

Capitolo 2

Teoria dei grafi

In questo secondo capitolo esporremo la teoria dei grafi, in particolare tutto ciò che sarà utile per descrivere un labirinto, tra cui la definizione di cammino, connessione, isomorfismo, il problema di planarità e gli algoritmi di visita di un grafo (per approfondimenti si veda [1]).

2.1 Generalità di un grafo

Definizione 2.1. Un *grafo* G è una coppia (V, E) di insiemi tale che $E \subseteq V \times V/\Sigma_2$, con Σ_2 gruppo delle permutazioni su due elementi.

Gli elementi di V vengono chiamati *vertici* (o *nodi*) del grafo G, mentre E è un sottoinsieme di coppie non ordinate di V dette *lati* (o *archi*).
Introduciamo ora la terminologia che utilizzeremo.
Siano $u, v \in V$, se $\exists e = \{u, v\} \in E$ allora si dice che u e v sono tra loro *adiacenti* e costituiscono gli estremi del lato e; al tempo stesso diremo che l'arco e è incidente ai vertici u e v.
Il *grado* (o *valenza*) di un vertice v è il numero degli archi incidenti ad esso; un vertice di grado 0 è detto *isolato*.
Un *cappio* è un arco con estremi coincidenti; un grafo è detto *semplice* se non contiene cappi o archi multipli (cioè se ogni coppia di vertici è collegata da al massimo un arco).

Convenzionalmente con le lettere n ed m si indica la cardinalità di V ed E, rispettivamente, ossia il numero di vertici e di archi del grafo: $n = |V|$ n viene chiamato *ordine* del grafo; entrambi gli insiemi possono essere finiti o infiniti. Data la natura del nostro problema considereremo solo grafi con n e m finiti.

Esempio 2.1. Un esempio di grafo è $G = (V, E)$ dove

$$V = \{v_1, v_2, v_3, v_4, v_5, v_6\} \quad E = \{e_1, e_2, e_3, e_4, e_5, e_6, e_7\}$$

con

$$e_1 = \{v_1, v_6\} \quad e_2 = \{v_1, v_2\} \quad e_3 = \{v_1, v_3\} \quad e_4 = \{v_3, v_4\}$$
$$e_5 = \{v_2, v_4\} \quad e_6 = \{v_4, v_5\} \quad e_7 = \{v_2, v_5\}.$$

Dato un grafo G, possiamo inoltre rappresentarlo sul piano. Il *disegno* di un grafo è una funzione f che assegna ad ogni vertice $v \in V$ un punto $f(v)$ del piano ed assegna ad ogni arco $e \in E$ una curva $f(e)$ continua avente per estremi i punti corrispondenti agli estremi di e, vedi Figura 2.1.
È bene non confondere la struttura astratta di grafo con il suo disegno, dal momento che la collocazione dei vertici del grafo sul piano è del tutto arbitraria, come anche lo è la curva con cui si rappresentano i lati: per cui uno stesso grafo può essere disegnato in modi diversi tra loro. Ad esempio in Figura 2.1 sono riportati due disegni differenti del grafo G dell'Esempio 2.1.

2.2 Cammini, cicli e connessione

Definizione 2.2. Un *cammino* P in un grafo $G = (V, E)$ è dato da una sequenza ordinata e alternata di archi e vertici di G, $P = (v_0, e_0, v_1, \ldots, e_{n-1}, v_n)$ in cui l'arco e_i è incidente ai vertici v_i e v_{i+1}. Inoltre la sequenza deve necessariamente cominciare e finire con due nodi.

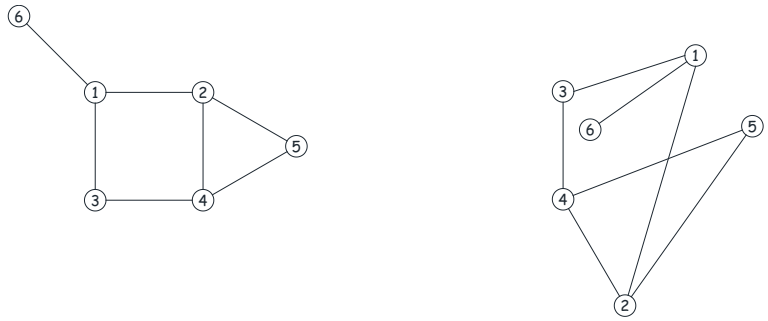


Figura 2.1: Due diverse rappresentazioni del grafo G dell'Esempio 2.1.

Un cammino da u a v è un cammino che ha come primo vertice della sequenza u e come ultimo v, questi sono detti gli *estremi* del cammino.
Si chiama *lunghezza* del cammino P da u a v il numero di archi in P. Un cammino avente u e v come estremi si dice *minimo* quando la sua lunghezza è la più piccola possibile al variare dei cammini aventi u e v come estremi. Si definisce *distanza* [1] tra due vertici la lunghezza del cammino minimo tra i due.
Un cammino è detto *semplice* se i vertici della sequenza sono tutti distinti, si veda Figura 2.2, ed è detto *chiuso* se gli estremi coincidono.

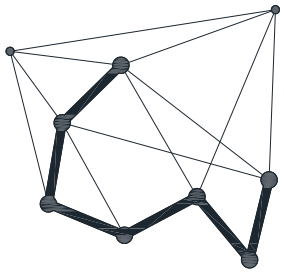


Figura 2.2: Un cammino semplice in grassetto, in un grafo G.

Definizione 2.3. Un *ciclo* è un cammino semplice e chiuso, si veda Figura 2.3. Un grafo in cui non sono presenti cicli è detto *aciclico*.

Andiamo ora a vedere un importante proprietà, quella di connessione tra vertici.

[1] La distanza tra due vertici è infinita se essi non sono collegati da un cammino.

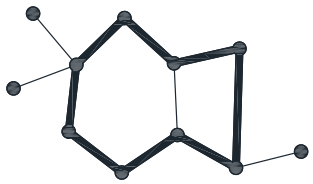


Figura 2.3: Un grafo con un ciclo evidenziato in grassetto.

Definizione 2.4. Sia $G = (V, E)$ un grafo, due vertici u e v si dicono *connessi* se esiste un cammino in G che li abbia come estremi. Se in un grafo G ogni coppia di vertici è connessa allora il grafo si dirà connesso.

Dunque la relazione di connessione determina sull'insieme dei vertici V, delle classi di equivalenza che costituiscono una partizione di V: tali classi sono denominate *componenti connesse*. In altre parole, i vertici che appartengono alla stessa componente connessa sono tutti mutuamente collegati da un cammino. Se due vertici appartengono a due componenti connesse distinte, allora vuol dire che non esiste nessun cammino che li collega.
Ci servirà in seguito definire un altro tipo di connessione

Definizione 2.5. Sia $G = (V, E)$ un grafo e $k \in \mathbb{N}$, il grafo G si dice *k-connesso* se $k < |V|$ e $G = (V \backslash X, E)$ è connesso per ogni $X \subset V$ con $|X| < k$

2.3 Alcune tipologie di grafi

Questa sezione è un'introduzione a tre importanti classi di grafi: gli alberi, i grafi orientati e i grafi pesati.
Si tratta di grafi che hanno molteplici interessi nelle applicazioni, per cui, nei prossimi capitoli, esporremo anche alcuni semplici ma importanti algoritmi che li riguardano.

Albero

Questo tipo di grafo è tra i più semplici, ma nonostante ciò ha una ricca struttura ed è utile in numerose applicazioni reali.

Definizione 2.6. Si chiama *albero* un grafo connesso e aciclico, si veda Figura 2.4. Un grafo privo di cicli (non necessariamente connesso) si chiama una *foresta*. Una foresta è dunque un grafo in cui ogni componente connessa è un albero.

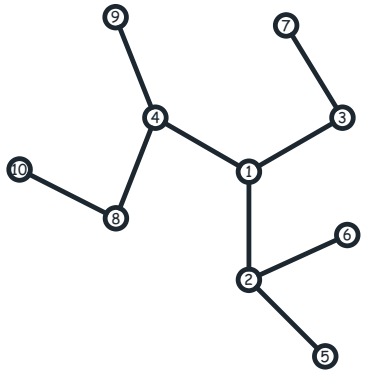


Figura 2.4: Un albero.

Iniziamo lo studio degli alberi enunciando i seguenti teoremi

Teorema 2.3.1. *Sia T un albero con n vertici. Allora esso possiede esattamente $n-1$ archi.*

Teorema 2.3.2. *Per ogni coppia di vertici in un albero T esiste un solo cammino che li collega.*

Nel paragrafo precedente abbiamo definito la lunghezza del cammino P che ha per estremi due vertici u e v in un grafo generico G e questa quantità cambia al variare del cammino scelto; mentre quest'ultimo teorema ci dice che in un albero, la lunghezza di P dipende solo dalla scelta dei due vertici e non più dal cammino, dato che ce n'è solo uno ad avere tali nodi come estremi.

Dato un albero, ci sarà utile in seguito mettere in risalto uno dei suoi vertici, diamo quindi la seguente

Definizione 2.7. Un *albero con radice* è una coppia (T, r) dove T è un albero e r uno dei suoi vertici chiamato *radice*, si veda Figura 2.5.

Mettendo in evidenza un qualunque nodo denominandolo radice, si instaura una gerarchia tra i vertici dell'albero in base alla loro distanza da esso.

Definizione 2.8. In un albero con radice, la *profondità* o *livello* di un vertice v è la sua distanza da r, cioè la lunghezza dell'unico cammino dalla radice a v (la radice ha livello 0). L'*altezza* di un albero con radice è la lunghezza del cammino più lungo dalla radice (o la massima profondità nell'albero).
Il vertice v si dice *predecessore* del vertice u (e v è *successore* di u) se u sta nell'unico cammino dalla radice a v; inoltre se u e v sono adiacenti si dice che v è *padre* di u (e u è *figlio* di v).

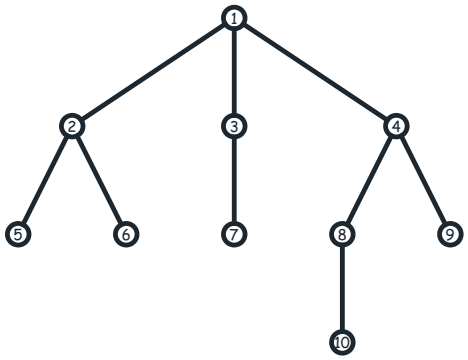


Figura 2.5: Albero rappresentato in Figura 2.4 con radice 1.

Grafi orientati

Nei capitoli precedenti abbiamo considerato esclusivamente grafi semplici: ovvero quelli in cui i lati sono definiti come coppie non ordinate di vertici distinti. Intendendo invece rappresentare una relazione binaria non necesariamente simmetrica, lo strumento è quello dei grafi orientati: ovvero grafi

in cui gli archi sono coppie ordinate di vertici. Di conseguenza ogni arco ha dunque un vertice iniziale ed un vertice finale, ed è quindi più propriamente rappresentato mediante una freccia. Più formalmente

Definizione 2.9. Si chiama *grafo orientato* G la tripla (V, E, f) tale che (V, E) è un grafo e f una funzione che associa ad ogni arco una coppia ordinata di vertici, si veda Figura 2.6.

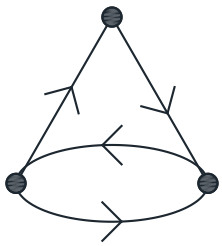


Figura 2.6: Un grafo orientato.

.

Grafo pesato

Fino ad ora, abbiamo rappresentato gli archi di un grafo come curve continue (o come frecce, nel caso di grafo orientato) di lunghezza arbitraria con il solo scopo di descrivere quali vertici essi collegano. Spesso però, nei problemi reali, si ha l'esigenza di associare un numero ad ogni arco $e = \{u, v\}$: esso viene chiamato *peso*.

Definizione 2.10. Un *grafo pesato* è un grafo G in cui ad ogni arco è associato un numero reale, solitamente positivo, tramite una funzione $p : E \longrightarrow \mathbb{R}$.

Naturalmente la nozione di peso dell'arco può essere applicata anche a grafi orientati, come in Figura 2.7.

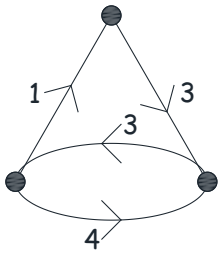


Figura 2.7: Un grafo pesato ed orientato.

2.4 Isomorfismo tra grafi

Definizione 2.11. Due grafi $G_1 = (V_1, E_1)$ e $G_2 = (V_2, E_2)$ si dicono *isomorfi* se esiste una corrispondenza biunivoca (isomorfismo) $\phi : V_1 \to V_2$ tale che

$$\{u, v\} \in E_1 \Leftrightarrow \{\phi(u), \phi(v)\} \in E_2.$$

In tal caso si scrive $G_1 \cong G_2$. Due coppie di grafi isomorfi sono riportati in Figura 2.8.

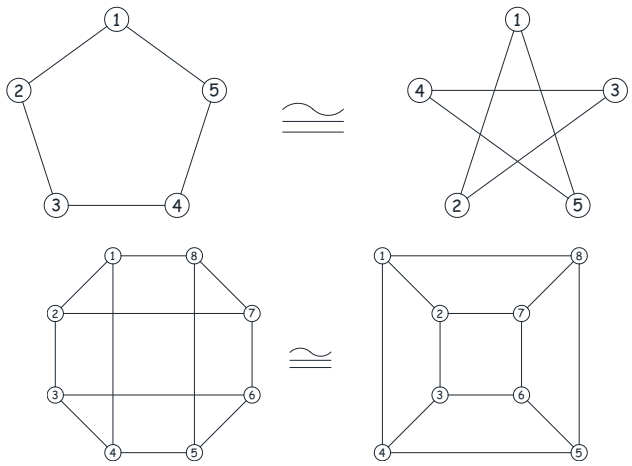


Figura 2.8: Esempi di grafi isomorfi.

Entro qualsiasi teoria matematica, gli oggetti isomorfi sono indistinguibili in termini di tale teoria, quindi lo scopo è quello di studiare le proprietà che rimangono inalterate sotto l'azione di un isomorfismo, cioè le proprietà

invarianti.
Di seguito, un elenco di proprietà invarianti:

- il numero dei vertici $n = |V_1|$ non cambia trasformando il grafo in un grafo isomorfo: l'isomorfismo è una corrispondenza biunivoca tra gli insiemi dei vertici dei due grafi e dunque la cardinalità dei due insiemi V_1 e V_2 deve essere la stessa;

- il numero degli archi $m = |E_1|$ del grafo: l'isomorfismo manda vertici adiacenti in vertici adiacenti ed è biunivoco, quindi anche il numero di lati deve rimanere invariato;

- il numero di componenti connesse del grafo: se G è costituito da k componenti connesse, allora ogni grafo isomorfo a G deve essere composto dallo stesso numero di componenti;

- il grado dei vertici: dato che l'isomorfismo conserva la proprietà di adiacenza di essi.

2.5 Immersioni

Definizione 2.12. Sia S una superficie nello spazio tridimensionale. In particolare si potrebbe trattare di un piano, una sfera o di un toro. Si chiama *immersione* di un grafo G su una superficie S un disegno di G tracciato su S senza che vi siano intersezioni tra gli archi eccetto che negli eventuali estremi comuni.

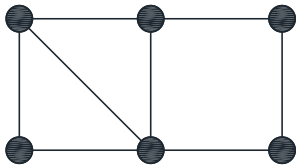
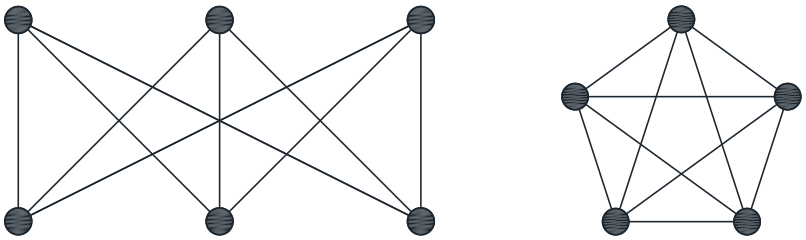


Figura 2.9: Un esempio di grafo immerso in un piano.

In Figura 2.9 un esempio di grafo immerso in un piano. Non tutti i grafi hanno un'immersione in una superficie fissata. Ad esempio non è possibile immergere i grafi K_5 o $K_{3,3}$, rappresentati in Figura 2.10, nel piano.



(a) $K_{3,3}$. (b) K_5.

Figura 2.10: Due grafi che non possiedono un'immersione nel piano.

Di particolare interesse sono i grafi che hanno un immersione sul piano.

Definizione 2.13. Un grafo G si definisce *planare* se è possibile immergerlo in un piano. Tale disegno è chiamato *immersione planare* di G. Un *grafo piano* è una particolare immersione planare di un grafo planare.

Definizione 2.14. Chiamiamo *faccia* di un grafo G immerso in una superficie S una componente connessa di $S - G$.

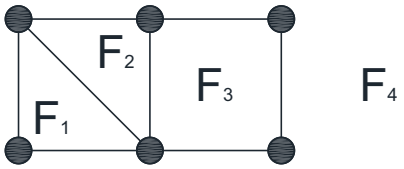


Figura 2.11: Il grafo in Figura 2.9, con le diverse facce etichettate.

Ad esempio, l'immersione planare di Figura 2.11 divide il piano in quattro regioni con frontiera gli archi di G: le facce triangolari F_1 e F_2, la faccia quadrata F_3 ed infine la faccia esterna F_4.
Enunciamo ora la Formula di Eulero che rappresenta uno dei più importanti

risultati per i grafi planari che mette in relazione vertici, archi e facce di un grafo piano.

Teorema 2.5.1 (Eulero)**.** *Se un grafo piano G connesso ha esattamente n vertici, m archi e f facce, allora*

$$n - m + f = 2$$

Il teorema di Eulero ha notevoli implicazioni nello studio dei grafi planari e ci porta a dire, per esempio, che tutte le immersioni nel piano di uno stesso grafo planare connesso G hanno lo stesso numero di facce.
La formula di Eulero può essere generalizzata al caso di grafi con k componenti connesse attraverso la formula $n - m + f = k + 1$.

Genere di un grafo

Il grafo K_5 in Figura 2.10 è detto *completo* perchè una qualsiasi coppia di vertici è collegata da un arco. Abbiamo detto che non c'e modo di disegnarlo nel piano senza avere intersezioni. Tuttavia notiamo che basta togliere un solo arco a K_5 per avere la planarità, questo significa che potremmo immergere K_5 in un toro, ossia una superficie orientabile, chiusa e connessa di genere 1. Introduciamo quindi la seguente definizione.

Definizione 2.15. Sia G un grafo, il *genere* $k \in \mathbb{N}$ di un grafo è il minimo genere tra quelli delle superfici orientabili chiuse in cui si può immergere G.

È facile vedere che ogni grafo piano si può immergere nella sfera e viceversa. Quindi i grafi planari hanno genere 0. Informalmente quindi il genere si può definire come il minimo numero di "manici" da aggiungere alla sfera per immergere G nella superficie così ottenuta.
In particolare K_5 ha genere 1, quindi per ottenere un immersione abbiamo bisogno di un piano con l'aggiunta di un manico in modo tale da rendere la superficie topologicamente equivalente ad un toro. Si parla perciò di grafi toroidali, si veda Figura 2.12.

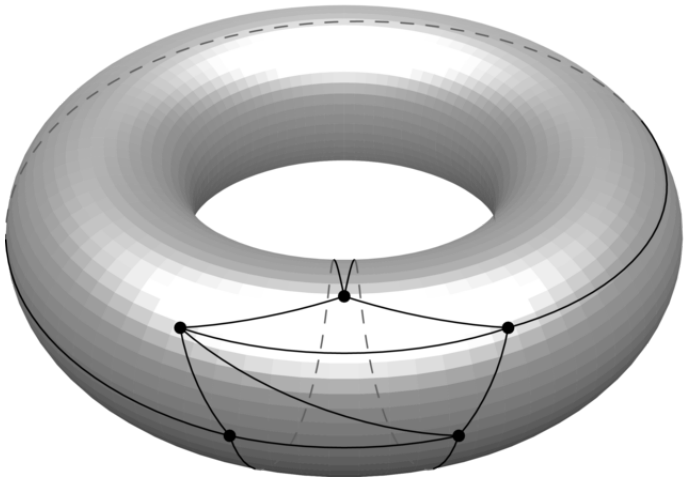


Figura 2.12: Il grafo K_5 immerso in un toro.

Caratterizzazione dei grafi planari

Un problema molto importante è, dato un grafo G qualunque, capire se questo possiede un'immersione in un piano, ovvero se è planare. Data l'importanza pratica dei grafi planari, occorre trovare un modo efficace per certificare la planarità. Diamo quindi le seguenti definizioni

Definizione 2.16. Sia dato un grafo $G = (V, E)$ ed un suo arco $e = (i, j) \in E$. L'operazione di *contrazione dell'arco* (i, j) consiste nel trasformare il grafo $G = (V, E)$ nel grafo $G' = (V', E')$ dove i nodi i e j sono sostituiti da un unico nodo s_{ij}, cioè

$$V' = (V/\{i, j\}) \cup \{s_{ij}\}$$

e gli archi sono gli stessi di E per quel che riguarda gli archi che non hanno come estremi i e j, l'arco $e = (i, j)$ viene soppresso e ogni arco che ha come uno dei due estremi i oppure j sostituisce tale estremo con il nodo s_{ij} (ad esempio l'arco (k, i) è sostituito con (k, s_{ij})).

Definizione 2.17. Sia dato un grafo G. Un grafo H è chiamato *minore* di un grafo G se H può essere ottenuto da G tramite un numero finito di operazioni di contrazione degli archi.

Andiamo ora ad enunciare un importante risultato che fornisce un test di planarità.

Teorema 2.5.2 (Kuratowski)**.** *Un grafo è planare se e solo se non contiene tra i suoi minori né K_5 né $K_{3,3}$.*

Tassellazione di una superficie

Si dicono *tassellazioni* i modi di ricoprire interamente una superficie con una o più figure geometriche senza sovrapposizioni o buchi. Tali figure geometriche, dette *tasselli*, sono spesso poligoni, regolari o no, ma possono anche avere lati curvilinei.
Più formalmente

Definizione 2.18. Sia S una superficie, si chiama tassellazione di S una collezione di regioni T_i topologicamente omeomorfe a poligoni del piano, ognuna delimitata da lati che sono curve continue tali che:

- $S = \bigcup_i T_i$
- per $i \neq j$, $T_i \cap T_j = \begin{cases} \emptyset & \text{oppure} \\ \text{un solo vertice} & \text{oppure} \\ \text{un lato e i suoi vertici estremi} \end{cases}$
- ogni lato è lato di esattamente due regioni distinte.

Va però tenuto presente che, dal punto di vista teorico, le tassellazioni di superfici non compatte, come il piano, hanno un numero infinito di facce mentre le realizzazioni concrete riguardano regioni limitate del piano, quindi data la natura del nostro problema considereremo soltanto tassellazioni con un numero finito di tasselli, come quella riportata in Figura 2.13.

Osserviamo che una tassellazione di una superficie è un immersione in termini di teoria dei grafi (cioè considerando i tasselli come poligoni, aventi

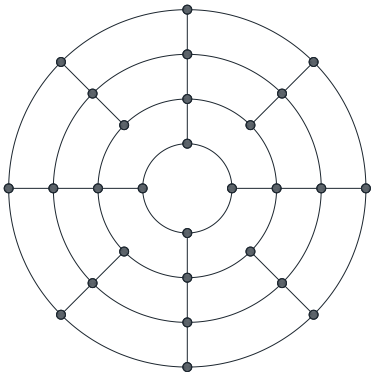


Figura 2.13: Esempio di una tassellazione di una porzione di piano.

i lati corrispondenti agli archi dell'immersione e i vertici corrispondenti ai nodi), ma non è vero il viceversa, cioè non tutte le immersioni sono tassellazioni, vedi Figura 2.14.

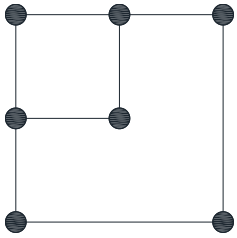


Figura 2.14: Un'immersione che non è una tassellazione.

Grafo duale

Data un'immersione di un grafo G su una superficie S è possibile costruire un grafo duale G^*, immerso in S, ad esso legato.

Definizione 2.19. Il grafo duale $G^* = (V^*, E^*)$ di un grafo immerso $G = (V, E)$ è un grafo che ha i vertici corrispondenti alle facce di G, mentre i suoi archi corrispondono a quelli di G nel seguente modo: se $e = (u, v)$ è un arco di G appartenente alle due facce X e Y, allora i vertici dell'arco duale

$e^* \in E^*$ sono x ed y associati alle facce X ed Y di G. Il grado del vertice $x \in V^*$ è uguale al numero di archi della faccia X di G.

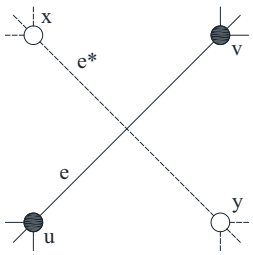


Figura 2.15: Un esempio di costruzione del duale.

La Figura 2.15 schematizza questo processo, mostrando la costruzione degli archi nel grafo duale, mentre in Figura 2.16 è riportato un esempio di grafo con il suo grafo duale.

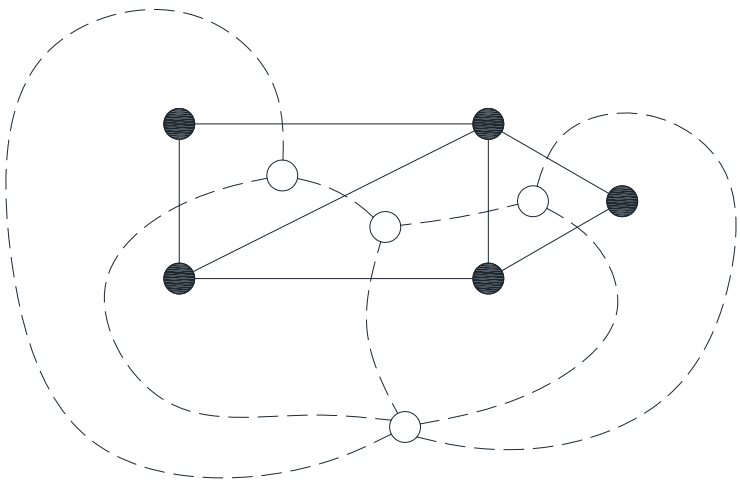


Figura 2.16: Il grafo duale, tratteggiato, di un grafo.

2.6 Visitare un grafo

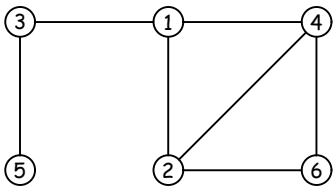
Visitare un grafo significa "esplorare" sistematicamente i nodi di un grafo attraverso i suoi archi. Questo ci è utile, ad esempio, per trovare un cammino da un dato vertice iniziale s ad un certo vertice v, oppure per esplorare tutti i vertici raggiungibili da s.
Andiamo ora a presentare due algoritmi di visita di un grafo, la visita in ampiezza (o BFS, dall'inglese, breadth-first search) e la visita in profondità (o DFS, cioè depth-first search).
Entrambi utilizzano due array, uno per determinare l'ordine di visita dei nodi utile al solo funzionamento dell'algoritmo, l'altro, chiamato output, servirà a stabilire quale vertice è già stato visitato per non visitarlo nuovamente, inoltre ci darà l'ordine finale di visita dei vertici.
In aggiunta per poter individuare i nodi scoperti o meno, entrambe le procedure colorano i vertici con tre colori: bianco, grigio, nero. Tutti i vertici hanno inizialmente colore bianco, quando un vertice è incontrato per la prima volta diventa grigio, metre quando tutti gli adiacenti di un nodo grigio sono stati visitati allora diventa nero (per approfondimenti si veda [2]).

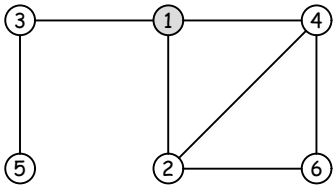
Breadth-First Search, BFS

La visita in ampiezza di un grafo, dato un vertice iniziale s chiamato sorgente, consiste nell'esplorazione sistematica di tutti i vertici raggiungibili da s in modo tale da visitare tutti i nodi che hanno distanza k prima di iniziare a visitare quelli che hanno distanza $k + 1$. In particolare utilizziamo una cosiddetta *queue*, o *coda*, per evidenziare la modalità di immagazzinamento dati (che sarà di tipo FIFO, *First in First out*) oltre all'array di output.
Parlando della colorazione dei vertici, in questo algoritmo se un nodo è bianco allora non è ancora stato visitato, se un nodo è grigio allora è stato messo nell'array di coda mentre se un nodo è nero vuol dire che è stato messo nell'output.
Possiamo schematizzare l'algoritmo nel seguente modo:

1. Mettere in coda (colorarlo di grigio) il nodo sorgente;
2. Finchè la coda non è vuota:
   (a) Prendere in esame un nodo grigio dalla coda (secondo la regola FIFO) e mettere in coda (colorandoli di grigio) tutti gli eventuali nodi adiacenti non ancora visitati (bianchi);
   (b) Togliere dalla coda il nodo appena esaminato ed inserirlo nell'output (colorandolo di nero).

Vediamo un esempio sul seguente grafo in cui la sorgente è il vertice etichettato con 1.



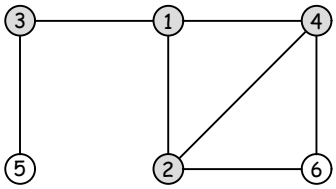


Coda

1

Mettiamo il nodo sorgente #1 in coda colorandolo di grigio.

OUTPUT

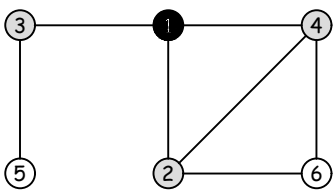


Coda

1 ←
2
3
4

Ora prendiamo in esame il vertice #1 e mettiamo in coda tutti i suoi adiacenti (bianchi) non ancora visitati (per semplicità in ordine numerico).

OUTPUT

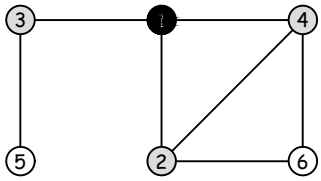


Coda

2
3
4

① Dato che del vertice #1 abbiamo visitato tutti i suoi adiacenti, questo esce dalla coda, diventa nero e viene messo nell'output.

OUTPUT : 1

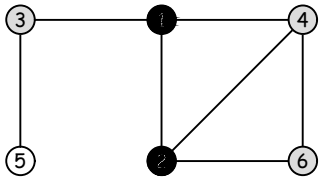


Coda

2 ←
3
4
6

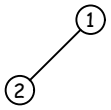
Ora prendiamo in esame il vertice #2 che ha adiacente e bianco solo il #6 che mettiamo in coda facendolo diventare quindi grigio.

OUTPUT : 1



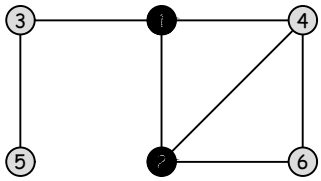
Coda

3
4
6



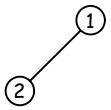
Come per il vertice #1, il #2 esce dalla coda, diventa nero e viene messo nell'output. Vediamo che comincia a costruirsi l'albero risultante avente i vertici presenti nell'output.

OUTPUT : 1, 2



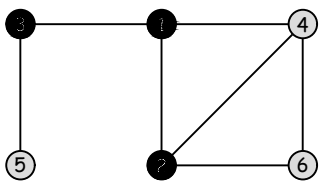
Coda

3 ←
4
6



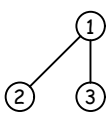
Ora visitiamo i nodi adiacenti al vertice #3, quindi mettiamo il #5 in coda.

OUTPUT : 1, 2



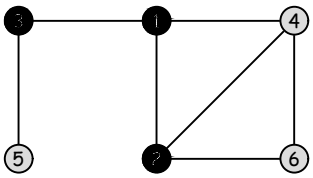
Coda

4
6
5

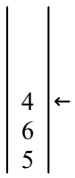


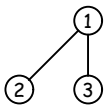
Il vertice #3 esce dalla coda, diventa nero e viene messo nell'output. Va anche rappresentato nell'albero risultante.

OUTPUT : 1, 2, 3



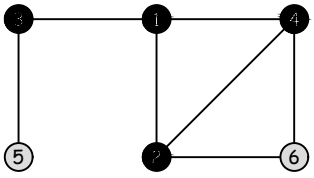
Coda



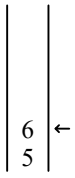


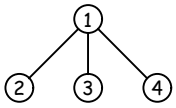
Il nodo #4 non ha vertici adiacenti da visitare, quindi non viene messo niente in coda. Mettiamo il vertice #4 nell'output facendolo diventare nero.

OUTPUT : 1, 2, 3



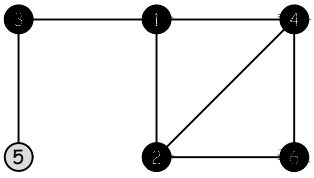
Coda



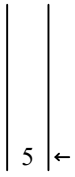


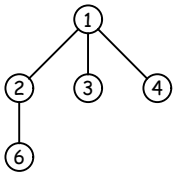
Lo stesso vale per il nodo #6.

OUTPUT : 1, 2, 3, 4



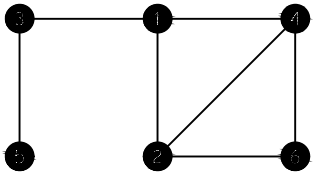
Coda



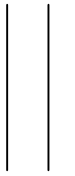


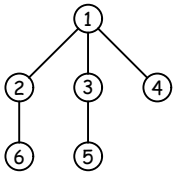
E per il #5.

OUTPUT : 1, 2, 3, 4, 6



Coda





La coda è vuota, quindi l'algoritmo termina dandomi come risultato l'albero a fianco.

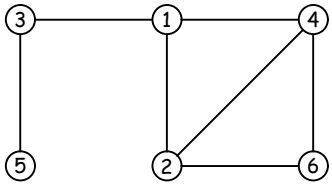
OUTPUT : 1, 2, 3, 4, 6, 5

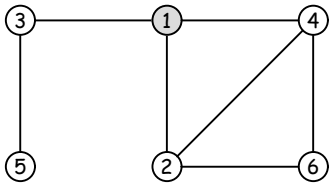
La particolarita di BFS è quella di andare a visitare il grafo per "livelli" dal nodo sorgente. Inoltre dato che un vertice viene visitato al massimo una volta, esso avrà un solo padre, ciò implica che l'algoritmo darà come risultato un albero che ha s come radice e che comprende tutti i vertici raggiungibili dal nodo sorgente. Inoltre per ogni $v \in V$ raggiungibile da s il cammino che ha come estremi s e v, nell'albero risultante, è minimo.

Depth-First Search, DFS

La visita in profondità di un grafo consiste nell'esplorazione sistematica di tutti i vertici andando in ogni istante il più possibile in profondità. I nodi vengono quindi visitati a partire dall'ultimo esplorato che abbia vertici adiacenti non ancora scoperti. Per realizzare ciò avremo bisogno di una modalita di immagazzinamento dati di tipo LIFO, *Last in First out*, utilizzando quindi una *stack*, o *pila*, oltre che di un array output.
In questo algoritmo invece i nodi colorati di grigio vengono messi direttamente nell'output e diventano neri quando li togliamo dalla pila (ciò implica che tutti gli adiacenti sono stati gia visitati).
Possiamo riassumere l'algoritmo nel modo seguente:

1. Mettere il nodo sorgente in pila e nell'array output (colorandolo di grigio);
2. Finche la pila non è vuota:
   (a) Se l'ultimo nodo messo della pila (grigio) ha vertici non ancora visitati (bianchi), scegliere a caso uno di essi e metterlo sia nella pila che nell'array output (facendolo diventare grigio);
   (b) Altrimenti se l'ultimo nodo messo nella pila è adiacente solo a vertici già visitati (grigi), toglierlo dalla pila e colorarlo di nero.

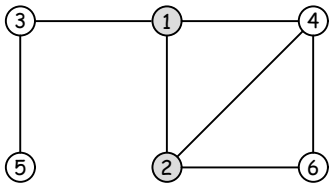




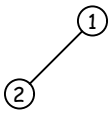
Pila

OUTPUT : 1

Mettiamo il nodo sorgente #1 in coda e nell'output facendolo diventare grigio.
Da qui parte la costruzione dell'albero con il solo vertice #1.

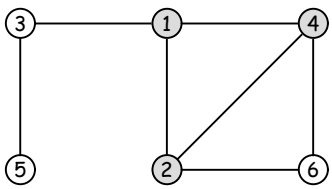


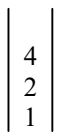
Pila



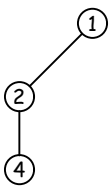
OUTPUT : 1, 2

Il nodo #1 ha vertici adiacenti bianchi, quindi ne prendo uno (ad esempio il #2, cioé andando in ordine numerico) lo metto in pila e nell'output. Il vertice #2 viene aggiunto all'albero.



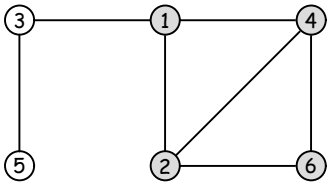


Pila



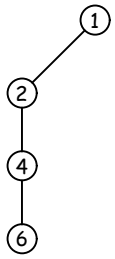
OUTPUT : 1, 2, 4

Il vertice #2 ha come adiacenti il #4 e il #6, scegliamo il #4 lo mettiamo in pila, nell'output e nell'albero.



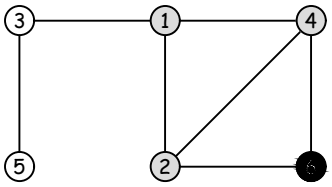
6
4
2
1

Pila



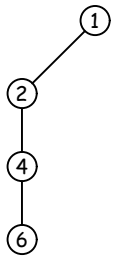
OUTPUT : 1, 2, 4, 6

Dal vertice #4 dobbiamo necessariamente visitare il #6.



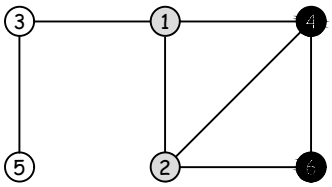
4
2
1

Pila



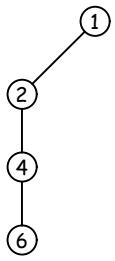
OUTPUT : 1, 2, 4, 6

Qui il vertice #6 non ha vertici adiacenti non ancora visitati quindi lo tolgo dalla pila colorandolo di nero.



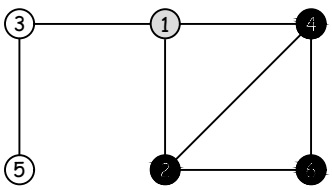
2
1

Pila



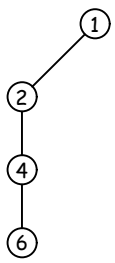
OUTPUT : 1, 2, 4, 6

Lo stesso vale per il vertice #4.



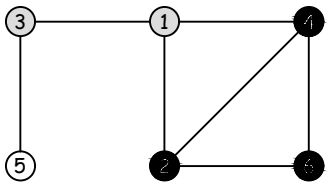
1

Pila

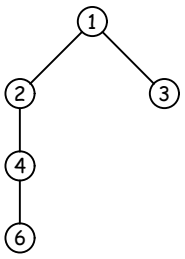


OUTPUT : 1, 2, 4, 6

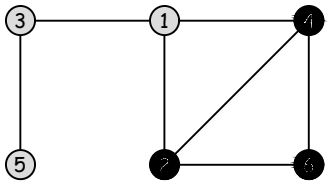
E per il vertice #2.



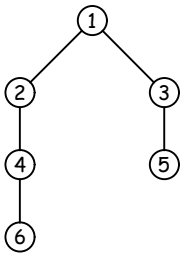
OUTPUT : 1, 2, 4, 6, 3



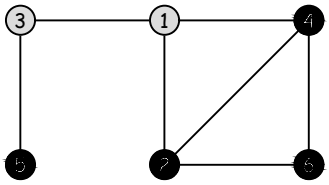
Ora considero il vertice #1 e mi muovo al nodo #3 facendolo diventare grigio, mettendolo in pila in output e nell'albero.



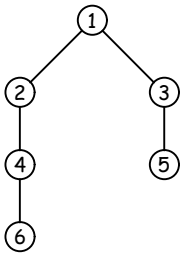
OUTPUT : 1, 2, 4, 6, 3, 5



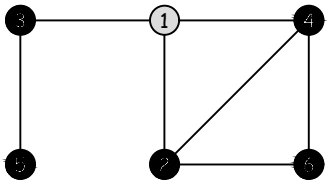
Lo stesso accade al vertice #5.



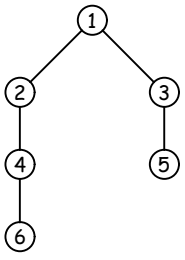
OUTPUT : 1, 2, 4, 6, 3, 5



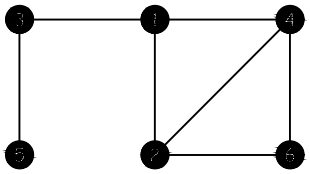
Quest'ultimo diventa nero e quindi lo tolgo dalla pila.

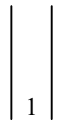


OUTPUT : 1, 2, 4, 6, 3, 5



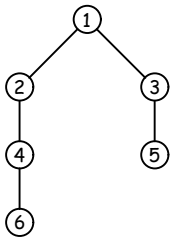
Lo stesso vale per il vertice #3.





Pila

OUTPUT : 1, 2, 4, 6, 3, 5



E per il vertice #1. La pila è vuota quindi l'algoritmo termina dando come risultato l'albero a fianco.

Come nel caso precedente i vertici vengono visitati al massimo una volta, quindi avranno un solo padre, perciò l'algoritmo crea un albero. A differenza di BFS, l'albero risultante non ci assicura di trovare il minimo cammino dalla sorgente s a qualunque altro vertice v raggiungibile da s, proprio per la natura di questo tipo di visita.

Capitolo 3

Teoria applicata ai labirinti

In queto capitolo andremo ad utilizzare la teoria dei grafi per descrivere, generare o risolvere un labirinto generico.

3.1 Grafo associato ad un labirinto

Andiamo ora a vedere come possiamo studiare, generare ed uscire da un generico labirinto utilizzando i grafi.
In questo paragrafo, quello che andremo a studiare è il grafo G che ci dice quante strade esistono all'interno di un labirinto e come sono collegate tra di loro. Andiamo a vedere come costruire tale grafo.
Prendiamo un generico labirinto, per modellizzare questa situazione in termini di teoria dei grafi dobbiamo individuare gli incroci e i vicoli ciechi presenti nel labirinto secondo la definizione data nel Capitolo 1. Essi, saranno i vertici del grafo che andremo a studiare (oltre al vertice di entrata e quello di uscita).
Per la costruzione degli archi di G dobbiamo andare a vedere quali vertici (corrispondenti agli incroci e ai vicoli ciechi) sono collegati da un percorso all'interno del labirinto. In questo modo possiamo associare ad ogni labirinto un grafo secondo le regole descritte in precedenza, vedi Figura 3.1.

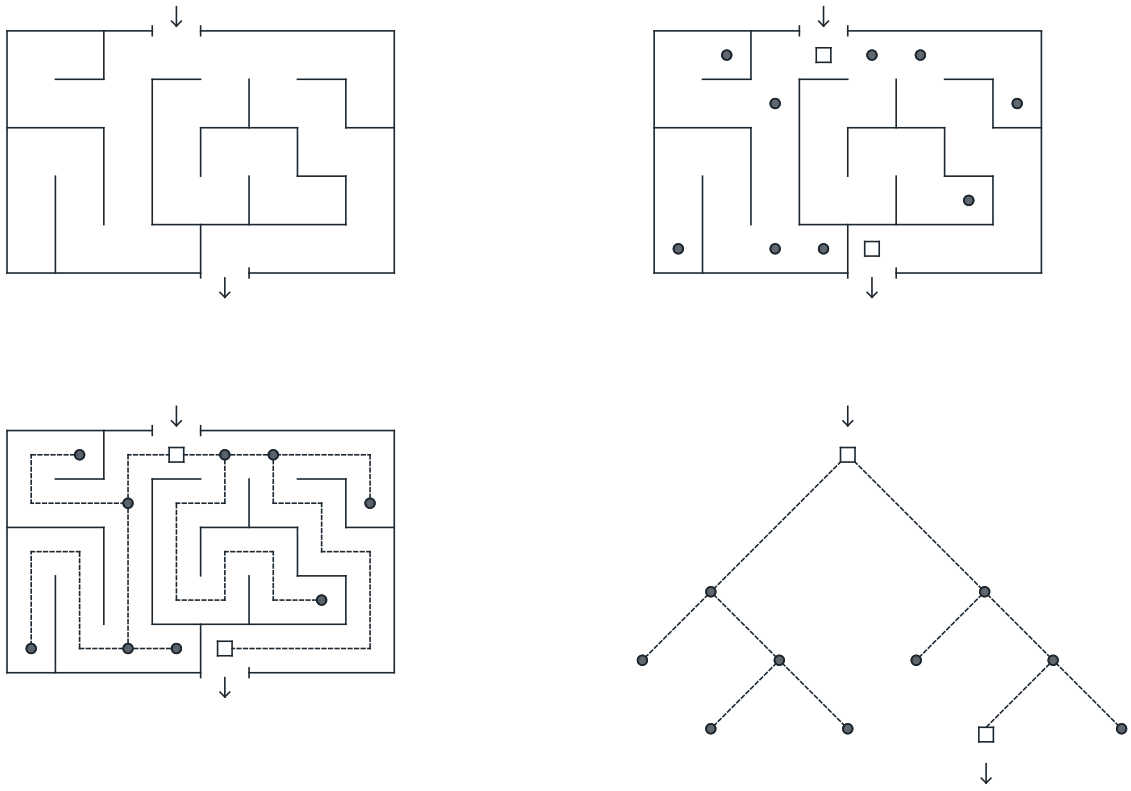


Figura 3.1: Un esempio di costruzione del grafo associato ad un labirinto.

Definizione 3.1. Due labirinti si dicono isomorfi se sono isomorfi i grafi ad essi associati.

In generale, una persona all'interno del labirinto non può avere una visione globale di esso. Ebbene, la forma topologica-geometrica non ha importanza né egli se ne rendo conto: per esempio i grafi-labirinti in Figura 3.2 sono per lui indistinguibili.
Chi percorre il labirinto al suo interno ha dunque percezione della sua struttura di grafo; chi lo vede dall'esterno può afferrare, oltre che la struttura di grafo, anche le sue proprietà geometriche. In particolare quando ci troviamo (senza una mappa) dentro ad un labirinto, ciò che facciamo è visitare il grafo G associato ad esso, e quindi se abbiamo due labirinti isomorfi, ci troveremmo praticamente a visitare il medesimo grafo.

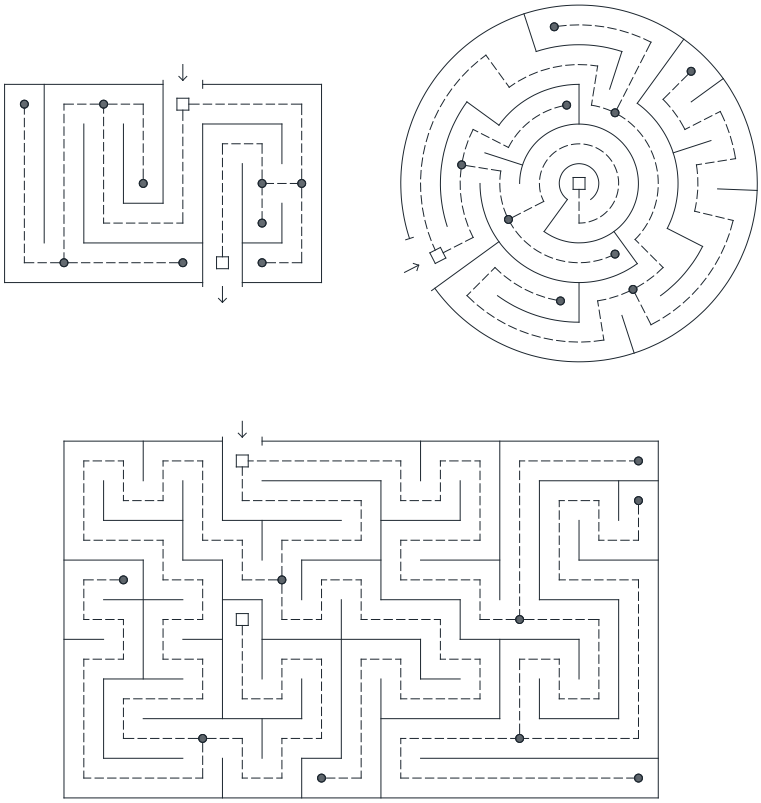


Figura 3.2: Esempi di labirinti isomorfi a quello di Figura 3.1.

Possiamo fare qualche osservazione: ad esempio non facciamo uso di grafi orientati, dato che ogni strada può essere percorsa in entrambi i sensi; non utilizziamo neanche grafi pesati perché ipotizziamo che tra le strade del labirinto non ce ne siano alcune eccessivamente dispendiose (in fatto di tempo o distanza) tali da ricorrere all'uso di questi ultimi. Poi, ad ogni labirinto sarà associato un grafo connesso, rendendo quindi accessibili tutte le zone del labirinto e naturalmente esso sarà di ordine finito.
Inoltre, per quanto riguarda il tipo di grafo, possiamo sia trovarci di fronte ad un albero, nel caso stessimo studiando un labirinto perfetto, ma anche di fronte a un grafo dove sono presenti cicli, nel caso di un labirinto braid.

3.2 Come generare un labirinto

Prima di dare algoritmi su come risolvere un labirinto, vediamo come possiamo generarne uno.
Esistono molti algoritmi per generare labirinti, tuttavia in questo paragrafo ci concetreremo su di un algoritmo in particolare che ci permette di generare labirinti perfetti. Tale scelta è dovuta al fatto che esso è molto versatile nel senso che ci permette di generare labirinti su ogni superficie, in ogni dimensione e per qualunque tassellazione; inoltre, togliendo determinati muri, è possibile trasformare un labirinto perfetto in un braid.

3.2.1 Algoritmo per generare

Recursive backtracker (DFS)

L'algoritmo di visita in profondità ci permette di generare un labirinto perfetto, andiamo a vedere come (per approfondimenti si veda [5]).
Conderiamo un grafo $G = (V, E)$, 2-connesso, che sia una tassellazione di una superficie S (o di una porzione di essa se consideriamo superfici non compatte) su cui vogliamo sviluppare il labirinto[1]; ora consideriamo il suo duale $G^* = (V^*, E^*)$ e, partendo da un nodo $v^* \in V^*$ qualunque, applichiamo ad esso l'algoritmo di visita in profondità.
Per far si che la visita DFS sul duale crei un labirinto a partire da G dobbiamo eliminare dal grafo G degli archi secondo la seguente regola: elimino $e \in E$ che separa le due facce X e Y se durante la visita di G^* ho esplorato $y \in V^*$ (o $x \in V^*$) tramite $e^* \in E^*$.
In altre parole, visitando in profondità G^*, si creano quelle che saranno le possibili strade da percorrere nel labirinto, mentre gli archi non eliminati dall'algoritmo formano quelli che saranno i muri.
Ad algoritmo terminato, ci troviamo di fronte a due diversi grafi, il grafo dei muri (corrispondente al grafo ottenuto da G eliminando determinati archi)

[1] La tassellazione usata è quella che poi caratterizzerà il labirinto che si andrà a costruire come visto nel Capitolo 1

ed il grafo dei percorsi (cioè l'albero risultante dall'algoritmo DFS applicato a G^*).

Nel caso stessimo considerando una tassellazione di una porzione di superficie non compatta, nella maggior parte dei casi, l'entrata e l'uscita in un labirinto sono poste esternamente, e quindi costruendo il duale del grafo si avrà che il nodo corrispondente alla faccia esterna sarà adiacente sia al nodo di entrata sia al nodo di uscita (in questo modo è teoricamente possibile risolvere il labirinto passando al di fuori di esso arrivando direttamente all'uscita). Per evitare questa situazione (e quindi rendere più difficile la risoluzione del labirinto) è opportuno togliere dal duale il nodo corrispondente alla faccia esterna con i relativi archi adiacenti.

Ora vediamo un esempio di come possiamo generare un labirinto da una tassellazione che per semplicità la prenderemo regolare, composta da 16 tasselli quadrati disposti su di un piano come in Figura 3.3. Per semplicità al posto dei nodi del duale andiamo a numerare le diverse facce della tassellazione considerando quindi che è possibile muoversi da una faccia all'altra solamente in direzione ortogonale.

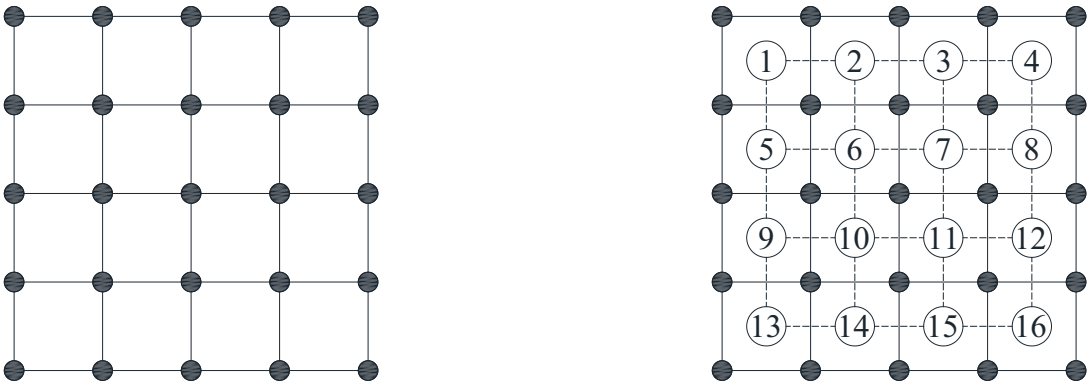


Figura 3.3: Un grafo e il suo duale con i vertici numerati, una volta eliminato il vertice corrispondente alla faccia esterna.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Pila

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Pila : 1

Coloriamo di rosso le celle già visitate. Partendo da una cella a caso (ad esempio la #1) e la mettiamo in pila.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Pila : 1

Ora cerchiamo le celle adiacenti alla #1 e le rappresentiamo in giallo, in questo caso coloriamo la #2 e la #5.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Pila : 1, 2

Abbiamo scelto arbitrariamente la #2, questa viene messa in pila e quindi colorata di rosso. Ora comincia la costruzione del labirinto creando un percorso tra la cella #1 e #2.

		3	4
5	6	7	8
9	10	11	12
13	14	15	16

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

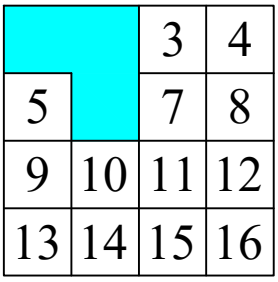
Pila : 1, 2

Ora dalla #2 evidenziamo le possibili scelte.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Pila : 1, 2, 6

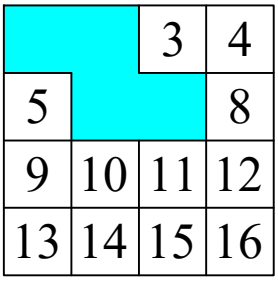
Abbiamo scelto la #6 e quindi messa in pila, intanto coloriamo le possibili scelte seguenti. Nel labirinto creiamo il percorso da #2 a #6.



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Pila : 1, 2, 6, 7

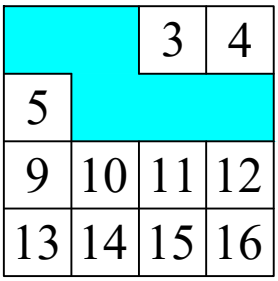
Abbiamo scelto la #7 e quindi messa in pila. Nel labirinto creiamo il percorso da #6 a #7.



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Pila : 1, 2, 6, 7, 8

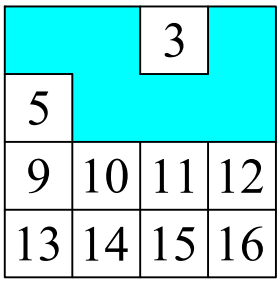
Abbiamo scelto la #8 e quindi messa in pila. Nel labirinto creiamo il percorso da #7 a #8.



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Pila : 1, 2, 6, 7, 8, 4

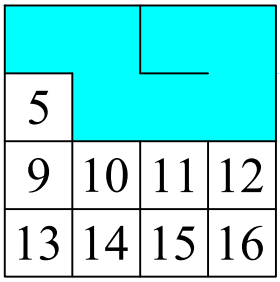
Abbiamo scelto la #4 e quindi messa in pila. Nel labirinto creiamo il percorso da #8 a #4.



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Pila : 1, 2, 6, 7, 8, 4, 3

Qui rimane solo la cella #3 da visitare e la mettiamo in pila. Nel labirinto creiamo il percorso da #4 a #3. Ora non abbiamo più scelte disponibili dalla cella #3.



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

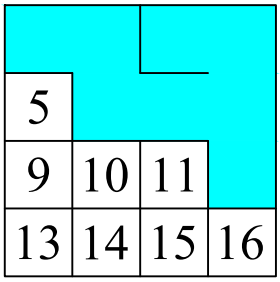
Pila : 1, 2, 6, 7, 8

Torniamo indietro eliminando dalla pila le celle da cui non possiamo muoverci (la #3 e la #4). Arriviamo quindi di nuovo alla #8.

1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Pila : 1, 2, 6, 7, 8, 12

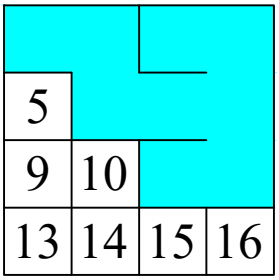
La #12 è l'unica scelta possibile, la mettiamo in pila. Nel labirinto creiamo il percorso da #8 a #12.



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Pila : 1, 2, 6, 7, 8, 12, 11

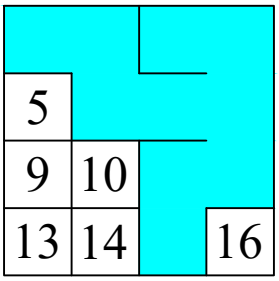
Abbiamo scelto la #11 e quindi messa in pila. Nel labirinto creiamo il percorso da #12 a #11.



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Pila : 1, 2, 6, 7, 8, 12, 11, 15

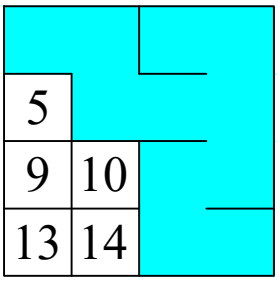
Abbiamo scelto la #15 e quindi messa in pila. Nel labirinto creiamo il percorso da #11 a #15.



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

Pila : 1, 2, 6, 7, 8, 12, 11, 15, 16

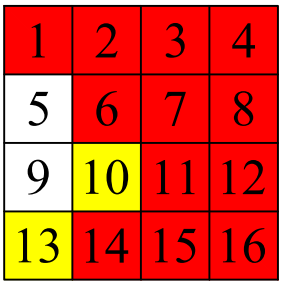
L'unica scelta possibile è la #16 e la mettiamo in pila. Nel labirinto creiamo il percorso da #15 a #16.



1	2	3	4
5	6	7	8
9	10	11	12
13	14	15	16

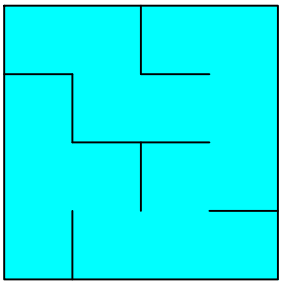
Pila : 1, 2, 6, 7, 8, 12, 11, 15

Torniamo indietro eliminando dalla pila la cella #16, trovandoci alla #15.



Pila : 1, 2, 6, 7, 8, 12, 11, 15, 14

Continuiamo il processo...



Quando l'algoritmo visita tutte le celle, mano a mano, dalla pila si eliminano tutte le celle una dopo l'altra e (dato che non è possibile fare più alcun movimento) questo conclude l'algoritmo.

3.3 Come uscire da un labirinto

Ora ci chiediamo, come possiamo tradurre il problema di uscire dal labirinto in termini di teoria dei grafi?
Abbiamo detto che per ogni tipo di labirinto possiamo associare un grafo, che ne descrive gli incroci e le strade in esso, perciò muoversi nel labirinto equivale a "muoversi nel grafo" ad esso associato. Qui ci vengono in aiuto gli algoritmi di visita di un grafo enunciati nel capitolo precedente che ci permettono di visitare in modo schematico il grafo (e quindi il labirinto) avendo come obbiettivo di trovare il nodo di uscita, tramite un cammino che parta dal nodo di entrata.
In particolare, in questo paragrafo vedremo due dei molteplici metodi per uscire da un labirinto utilizzando i grafi.

3.3.1 Algoritmi di risoluzione

Esistono innumerevoli algoritmi che permettono di trovare l'uscita da un labirinto, molti dei quali sono "informati", ciò vuol dire che chiunque voglia risolvere l'enigma (che sia una persona o un calcolatore) conosce a priori la struttura del labirinto (possiede una mappa o ce l'ha in memoria).
In questa tesi non ci occuperemo di questi, ma piuttosto andremo ad analizzare i principali algoritmi "non informati" che modellizzano la situazione in cui nella realtà generalmente si trova una persona dentro ad un labirinto.

Wall follower

Uno dei metodi più semplici per uscire da un labirinto è, come dice il nome, "seguire il muro". Sta a significare di dover appoggiare una mano ad un muro qualsiasi e proseguire il viaggio lungo tutto il labirinto senza mai staccarla, ciò equivale a dover girare sempre dalla stessa parte, ad esempio a destra, ad ogni incrocio che ci si presenta; se girassimo a sinistra si avrebbe la stessa identica situazione, l'importante è essere coerenti una volta effettuata la scelta (si veda Figura 3.4). In termini di teoria dei grafi questo proce-

dimento equivale a dover esplorare il grafo associato al labirinto andando a visitare, ad ogni iterazione, il vertice adiacente situato sulla destra (o sulla sinistra).
A discapito della sua semplicità, questo procedimento non funziona per tutti i tipi di labirinto, ad esempio quando l'uscita (o l'obiettivo) si trova all'interno circondato da un "isola" cioè circondato da muri non connessi con il muro esterno; in particolare per i labirinti perfetti questo metodo funziona sempre non avendo cicli al loro interno, ma non conoscendo a priori la sua struttura può essere un metodo fallimentare ed è per questo motivo che non è il più utilizzato.

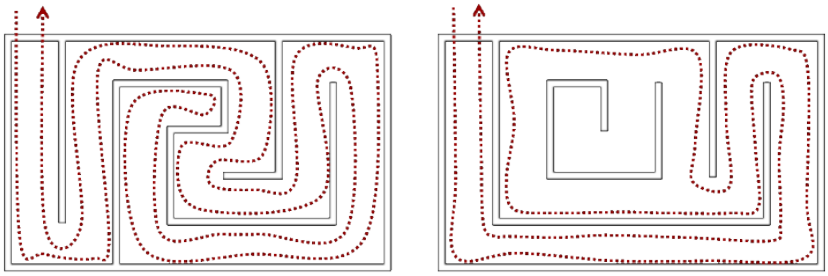


Figura 3.4: Esempi di risoluzione con wall follower.

Algoritmo di Trémaux

L'algoritmo di Trémaux, garantisce di trovare una via d'uscita da un qualunque tipo di labirinto. Per il corretto funzionamento di questo metodo bisogna avere un oggetto utile a segnare le vie intraprese, sapendo che non è concesso percorrere una strada già segnata due volte[2].
Chiamiamo *nuovo incrocio*, un incrocio da cui partono solo strade prive di alcun segno (un incrocio privo di segni), mentre un *vecchio incrocio* è un incrocio in cui, tra le strade a lui adiacenti, ve n'è almeno una segnata (un

[2]Per situazioni reali è bene segnare il percorso all'inizio e alla fine della strada in modo da rendelo visibile sia dall'incrocio di partenza sia da quello di arrivo

incrocio segnato almeno una volta).
Andiamo ora a descrivere l'algoritmo

1. Se si arriva ad un nuovo incriocio, scegliere arbitrariamente una strada priva di segno;

2. Altrimenti se si arriva ad un vecchio incrocio

   (a) Se possibile tornare indietro;

   (b) Altrimenti scegliere arbitrariamente una delle rimanenti strade con il minor numero di segni.

Chiaramente l'algoritmo si interrompe quando è stata trovata l'uscita, inoltre se non ce n'è una, ci conduce al nodo di entrata dopo aver segnato due volte ogni percorso. Possiamo dimostrare che l'algoritmo funziona tramite questo lemma

Lemma 3.3.1. *In qualunque istante di visita, prima di arrivare all'incrocio successivo o dopo essere partiti dal precedente, l'incrocio da cui siamo partiti entrando nel labirinto possiede un numero dispari di segni mentre ogni altro incrocio ne possiede un numero pari.*

Dimostrazione. Se arriviamo ad un nuovo incrocio l'algoritmo ci dice di prendere una qualunque altra strada che parta da lì, quindi lascieremo l'incrocio con due segni su strade diverse a lui adiacenti, mentre se arriviamo ad un vecchio incrocio (questo avrà un numero pari di segni, almeno due altrimenti sarebbe considerato nuovo) l'algoritmo ci dice di tornare indietro e quindi aggiungergli altri due segni o se non è possibile di continuare il viaggio lasciando in ogni caso l'incrocio con due segni in più.
Ci allontaneremo dall'incrocio dal quale siamo partiti lasciando un solo segno e se durante la visita ci ricapiteremo si andranno ad aggiungere altri due segni lasciandone in ogni caso un numero dispari. □

Per la sua costruzione, l'algoritmo può interrompersi solo nel caso arrivassimo ad un vecchio incrocio, da cui partono strade già segnate due volte,

tramite una strada segnata una volta in precedenza, ci troveremmo quindi in un incrocio dove non è possibile percorrere alcuna strada. Ecco, questo non può accadere perché la situazione descritta in precedenza implica che l'incrocio considerato (eccetto quello di partenza) abbia un numero dispari di segni e questo è in disaccordo con il lemma.
Vediamo un esempio dell'esecuzione di questo algoritmo su di un labirinto esistente, quello di Hampton court, vedi Figura 3.5 situato a Londra.

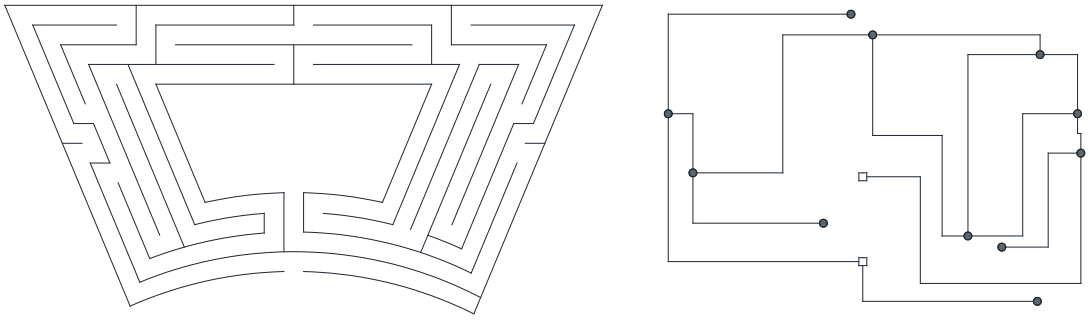
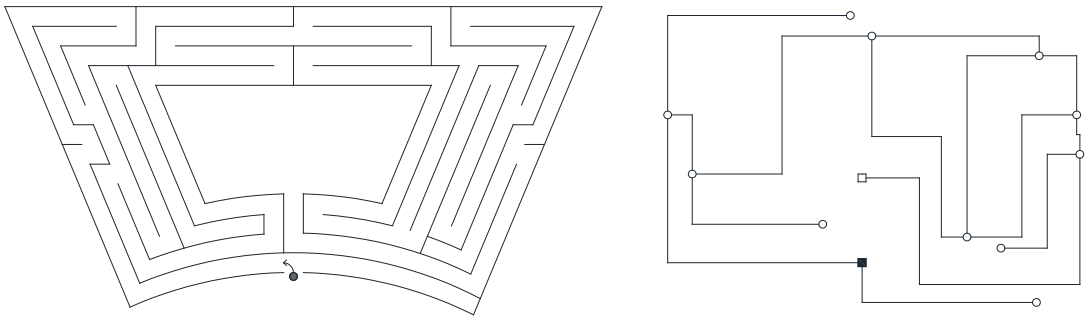
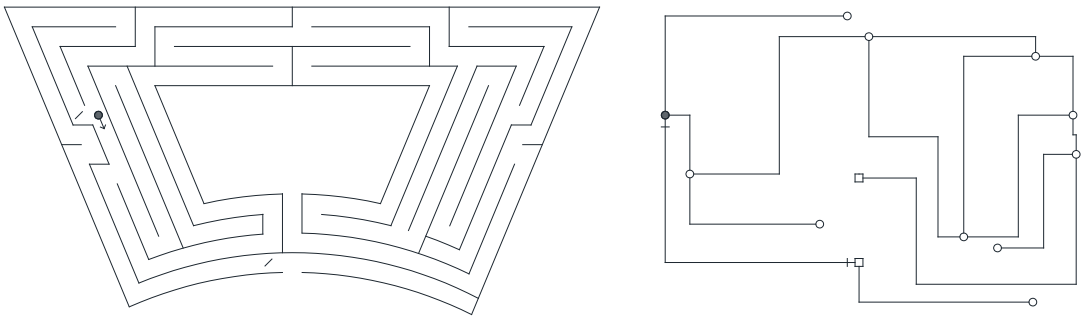


Figura 3.5: Il labirinto di Hampton Court ed il grafo ad esso associato.

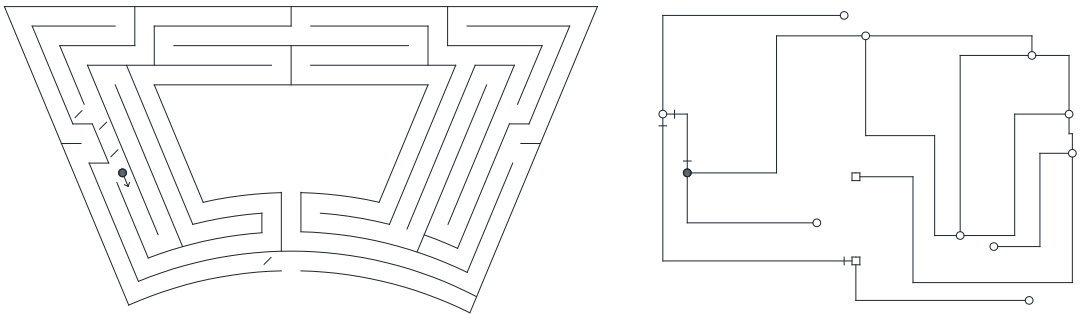
Con un cerchietto nero rappresenteremo la nostra posizione all'interno del labirinto e muovendoci andremo a segnare, in questo caso con dei piccoli segmenti, il percorso intrapreso.



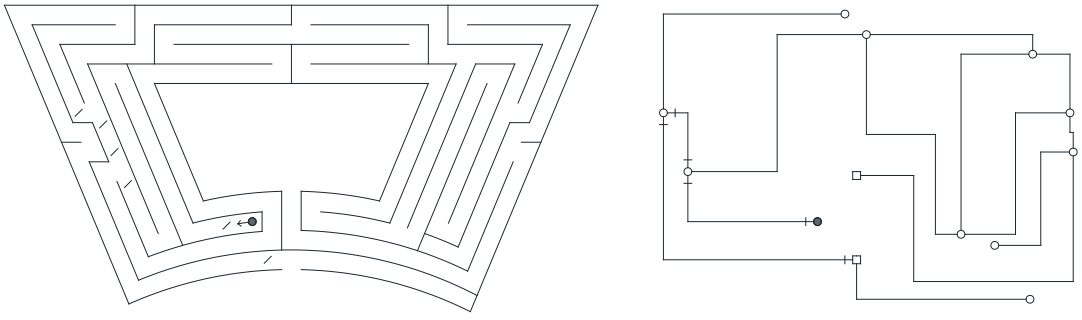
Siamo entrati nel labirinto, abbiamo scelto di svoltare arbitrariamente a sinistra, indicandolo con una freccia, quindi percorriamo la strada fino al prossimo incrocio.



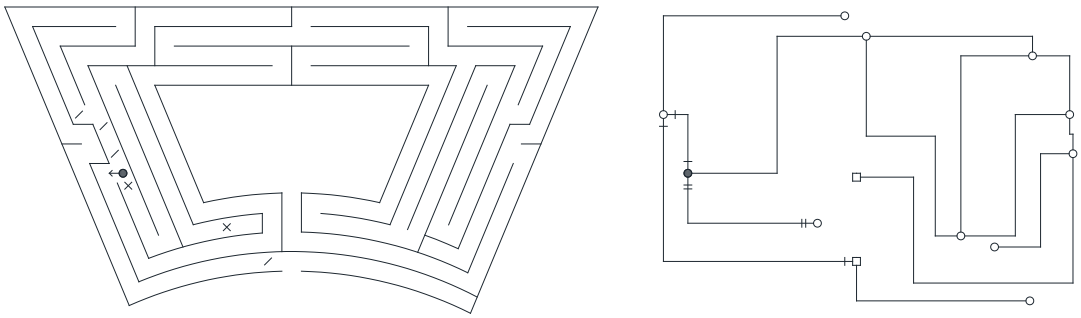
Arrivati all'incrocio successivo e avendo segnato entrambi i capi del percorso intrapreso, scegliamo arbitrariamente una strada tra quelle prive di segno arrivando fino al prossimo incrocio.



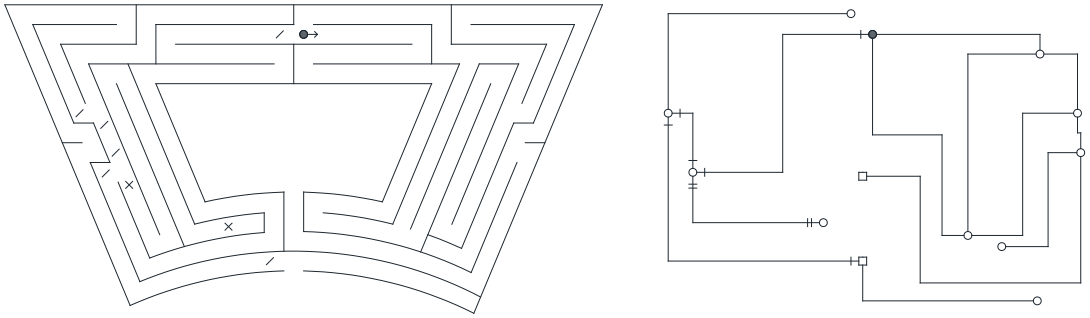
Come prima abbiamo segnato la strada intrapresa e proseguiamo per un percorso privo di segno.



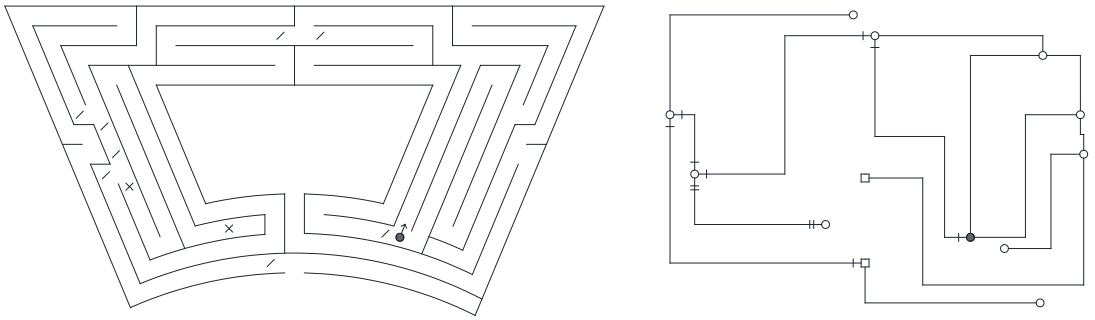
Siamo in un vicolo cieco, l'algoritmo ci dice di prendere l'unica strada possibile da qui, cioè di tornare indietro fino all'incrocio precedente.



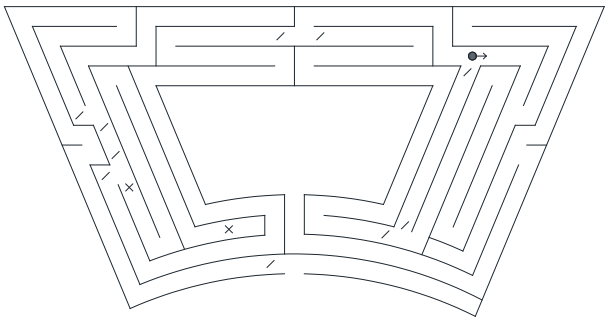
Abbiamo quindi segnato due volte la strada che conduce ad un vicolo cieco, quindi non dovremmo più percorrerlo d'ora in poi. Siamo ad un vecchio incrocio, dato che non è possibile tornare indietro, l'algoritmo dice di prendere una strada qualsiasi con il minor numero di segni, quindi prendiamo l'unico percorso privo di segni.

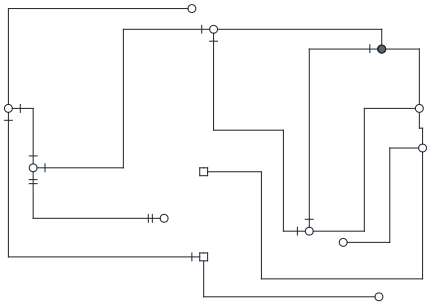


Arrivati ad un nuovo incriocio sciegliamo arbitrariamente una strada da percorrere.

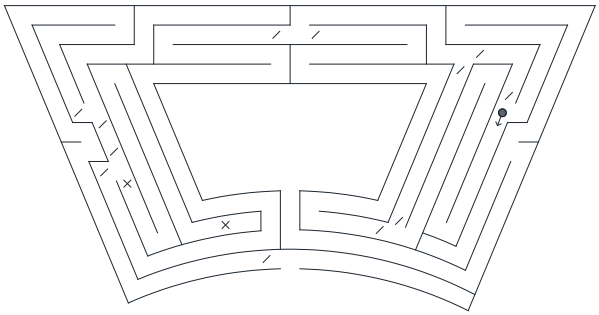


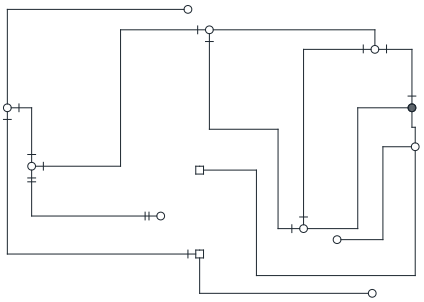
Come prima, proseguiamo per un percorso privo di segni.



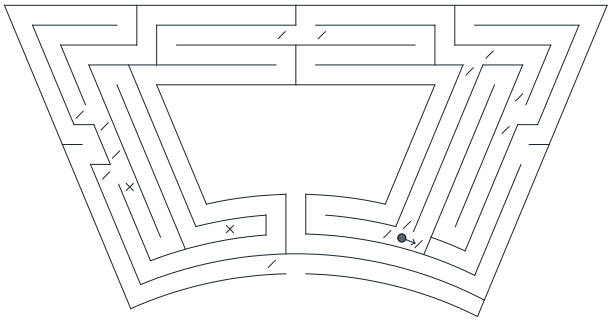


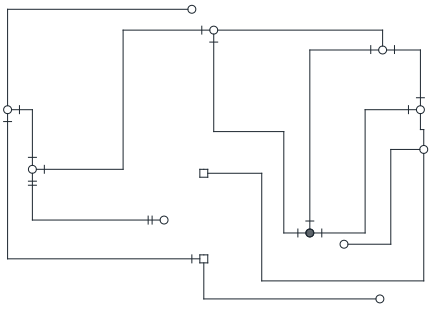
Proseguiamo ancora.



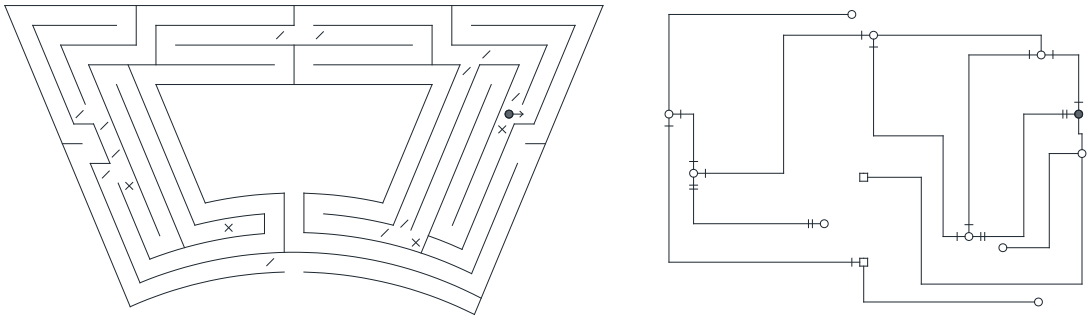


E ancora.

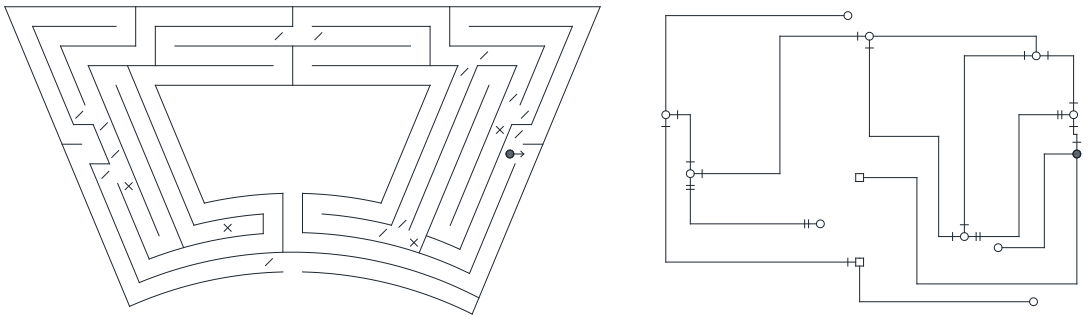




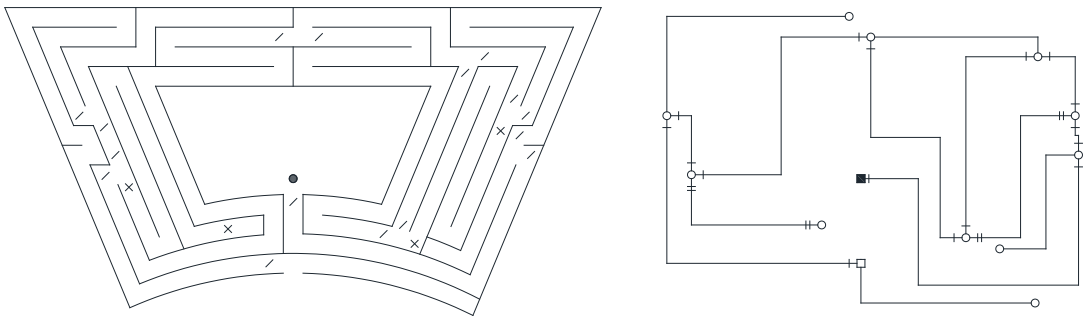
Ora siamo arrivati ad un vecchio incrocio, dato che il percorso che ci ha portato ad esso ha un solo segno, l'algoritmo ci dice di tornare indietro fino all'incrocio precedente.



Arrivati al precedente incrocio, segnamo quindi per la seconda volta la strada appena percorsa e continuiamo la visita percorrendo l'unica strada priva di segno.



Siamo ora ad un nuovo incrocio, quindi scegliamo arbitrariamente un percorso tra i due disponibili privi di segno.



Siamo arrivati al centro del labirinto, l'algoritmo si interrompe.

Considerando il grafo G associato al labirinto, l'algoritmo di Trémaux su di esso non fa altro che visitare G in profondità (si veda Sezione 2.6) andando a segnare, mano a mano, gli archi del grafo. Se su G si seguono le regole descritte in precedenza per il corretto funzionamento dell'algoritmo, allora questo ci condurrà al nodo di uscita (nell'esempio precedente, al centro del labirinto).

Capitolo 4

Problema di cammino minimo e applicazioni

In quest'ultimo capitolo andremo ad estendere il concetto di labirinto ad un qualcosa di più generale del semplice rompicapo che possiamo trovare in un giornalino di enigmistica.
L'idea del "labirinto" può essere attribuito a molteplici situazioni, come ad esempio, alla rete stradale di una città, a giochi come gli scacchi o il cubo di rubik, al web; e di conseguenza ad ognuna possiamo associare un grafo G: per quanto riguarda le reti stradali, i vertici di G sono gli incroci e gli archi le strade, per i giochi, i nodi sono le possibili configurazioni e i lati sono le "mosse" per passare da una configurazione all'altra, mentre per il web, i nodi rappresentano i siti internet e i lati i cosiddetti "links" che portano da una pagina web all'altra.

Quello che osserviamo da questi esempi è che il grafo associato ad ognuno di loro risulta molto più vasto di quello di un semplice labirinto di un parco giochi, (vedi Figura 4.1): basti pensare che il grafo associato al cubo di rubik $3 \times 3 \times 3$ ha più di $43 \cdot 10^{18}$ vertici, di cui solo uno corrisponde alla configurazione corretta; quindi gli algoritmi di visita di un grafo che abbiamo dato, seppur funzionanti, non sarebbero efficenti.
Questo ci porta a sollevare un altro tipo di problema, dato un grafo G, come

Figura 4.1: La rete aerea mondiale. Ogni vertice rappresenta un aeroporto e ogni arco una tratta aerea tra due di essi. In totale questa rete contiene approssimativamente 4000 aeroporti e 25000 archi.

possiamo trovare un cammino tra due vertici con lunghezza minima[1]?
Il problema della ricerca del cammino minimo può essere definito sia su grafi orientati che su grafi non orientati. Esso può essere così formalizzato: dato un grafo pesato $G = (E, V, f)$, e dati inoltre due vertici dististi $u_1, u_n \in V$ trovare un cammino avente i suddetti vertici come estremi che minimizzi la somma

$$\sum_{i=1}^{n-1} f(e_{u_i,u_{i+1}})$$

con $e_{u_i,u_{i+1}}$ arco incidente ai vertici u_i e u_{i+1}. Naturalmente possiamo applicarlo anche a grafi non pesati ponendo come peso di ogni arco il valore 1.
Di questo problema esistono alcune varianti in cui, partendo da un dato vertice, può essere richiesto di trovare i cammini minimi verso tutti gli altri vertici; oppure trovare i cammini minimi per ogni coppia di vertici del grafo. Un problema simile è quello del "commesso viaggiatore" (i cosiddetti problemi di *routing*, che possono moldellizzare la situazione di consegna della posta o della merce, o lo smaltimento dei rifiuti) in cui si cerca il percorso più

[1]Peso minimo se lavoriamo con grafi pesati.

breve che attraversi tutti i vertici del grafo una sola volta, per poi ritornare al punto di partenza.

4.1 Algoritmo di Dijkstra

Il metodo di Dijkstra è un algoritmo di visita che permette di trovare il cammino minimo tra due vertici di un grafo, anzi permette di trovare il cammino minimo da un dato vertice (sorgente) a tutti gli altri vertici del grafo. Esso può essere applicato sia a grafi orientati che pesati, tuttavia, è necessario che gli archi del grafo abbiano peso non negativo.
In questo algoritmo, l'insieme dei nodi del grafo è diviso in tre parti: l'insieme dei nodi *visitati* V, l'insieme dei nodi di *frontiera* F che comprende i successori di z, cioè tutti i nodi raggiungibili lungo un arco uscente dal vertice z in esame e l'insieme dei nodi *sconosciuti*, che sono ancora da esaminare.
Per ogni nodo z, l'algoritmo tiene traccia di un valore d_z, inizialmente posto uguale a ∞ (che rappresenta il peso del cammino dal nodo sorgente a z) e di un nodo u_z (che nell'albero risultante rappresenta il padre e quindi ci dice come costruirlo).
Proprio per il fatto di voler trovare il minimo cammino, i nodi possono essere visitati dall'algoritmo più di una volta (a differenza di BFS e DFS) in modo da poter, ad ogni iterazione, aggiornare eventualmente il valore d_v.
L'algoritmo consiste nel ripetere il seguente passo: si prende dall'insieme F un qualunque nodo z con d_z minimo, si sposta z da F a V, si spostano tutti i successori di z sconosciuti in F e per ogni successore w di z si aggiornano i valori d_w e u_w. L'aggiornamento viene effettuato con la regola

$$d_w \leftarrow \min\{d_w, d_z + p_{z,w}\}$$

dove $p_{z,w}$ è il peso dell'arco che collega z a w. Se il valore di d_w è stato effettivamente modificato, allora u_w viene posto uguale a z.
Se cerchiamo il minimo cammino dal nodo sorgente s ad un nodo fissato, l'algoritmo parte con $V = \emptyset$, $F = \{s\}$, $d_s = 0$ e prosegue finché non viene

visitato il nodo fissato, o finché $F = \emptyset$. Vediamo un esempio applicato al grafo in Figura 4.2.

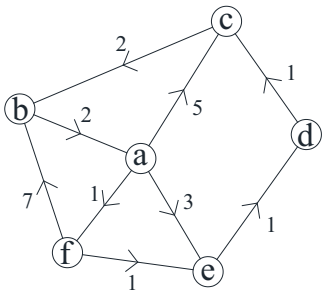
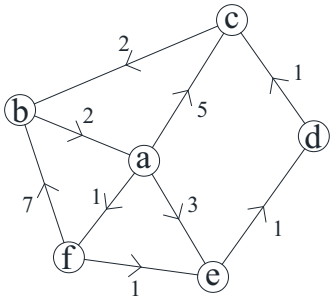


Figura 4.2: Un grafo orientato e pesato su cui applicare l'algoritmo Dijkstra.

Scegliamo quindi un nodo sorgente (nel nostro caso abbiamo scelto arbitrariamente il nodo a), inizialmente tutti gli altri vertici sono posti a distanza infinita, mentre u_z è ancora sconosciuto per ogni vertice.

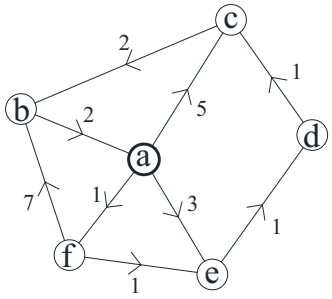


V = Ø

F = {a}

	a	b	c	d	e	f
u_z						
d_z	0	∞	∞	∞	∞	∞

Ora visitiamo a mettendo nell'insieme frontiera F tutti i suoi successori, quindi vediamo che le distanze d_z si aggiornano per i nodi c, f ed e, di conseguenza poniamo u_z uguale ad a per $z = c, f, e$.

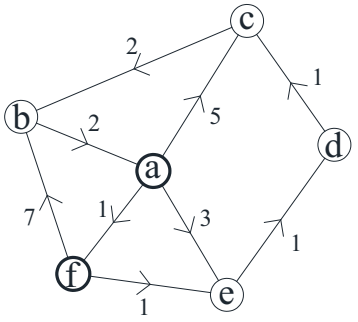


V = {a}

F = {c, e, f}

	a	b	c	d	e	f
u_z			a		a	a
d_z	0	∞	5	∞	3	1

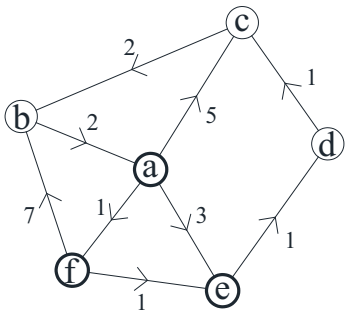
Visto che tra i nodi di F, f è quello con d_z minore proseguiamo la visita da esso, quindi lo spostiamo in V e mettiamo il suo successore b in F. Aggiorniamo le distanze, in particolare $d_e = d_f + p_{f,e}$ (con $p_{f,e}$ il peso dell'arco che va da e a f) e d_b e poniamo u_e e u_b uguale ad f.



V = {a, f}
F = {c, e, b}

	a	b	c	d	e	f
u_z		f	a		f	a
d_z	0	8	5	∞	2	1

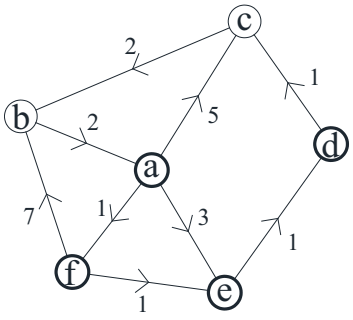
Proseguiamo la visita da e, lo mettiamo in V e il suo successore d in F ed aggiorniamo la sua distanza $d_d = d_e + p_{e,d} = 3$ e $u_d = e$.



V = {a, f, e}
F = {c, b, d}

	a	b	c	d	e	f
u_z		f	a	e	f	a
d_z	0	8	5	3	2	1

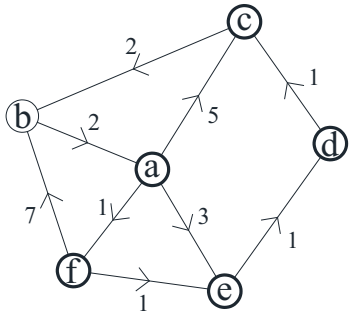
Proseguiamo dal nodo d e aggiorniamo d_c dato che la distanza è minore passando per il nodo d, quindi poniamo $u_c = d$



V = {a, f, e, d}
F = {c, b}

	a	b	c	d	e	f
u_z		f	d	e	f	a
d_z	0	8	4	3	2	1

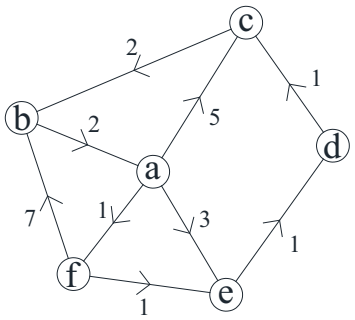
Ora visitiamo il nodo c, quindi aggiorniamo $d_b = d_c + p_{c,b} = 6$ e $u_b = c$.



V = {a, f, e, d, c}
F = {b}

	a	b	c	d	e	f
u_z		c	d	e	f	a
d_z	0	6	4	3	2	1

Abbiamo visitato tutti i vertici, $F = \emptyset$ questo conclude l'algoritmo. Nella tabella a fianco possiamo quindi vedere al variare di z tra i vertici del grafo il peso d_z del cammino minimo che parte dal nodo sorgente a e finisce in z e il vertice u_z, cioè il padre di z.



V = {a, f, e, d, c, b}
F = Ø

	a	b	c	d	e	f
u_z		c	d	e	f	a
d_z	0	6	4	3	2	1

Concludiamo l'esempio mostrando che l'algoritmo restituisce come output l'albero dei cammini minimi con sorgente a, questo perché l'ultimo aggiornamento dei vertici u_z ci dice qual è il padre di z tale da rendere il cammino da a a z minimo (vedi Figura 4.3).

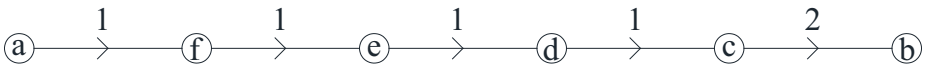


Figura 4.3: L'albero dei minimi cammini dell'esempio precedente.

Bibliografia

[1] Reinhard D., *Graph theory*, Heidelberg, Springer, 2016.

[2] Thomas H., Cormen, Charles E. Leiserson, Ronald L. Rivest, Clifford Stein, *Introduction to algorithm*, Massachusetts, MIT Press, 2009.

[3] D'Amore B., *Arte e matematica*, Bari, Dedalo, 2015.

[4] www.astrolog.org/labyrnth/algrithm.htm.

[5] www.datagenetics.com/blog/november22015/index.html.

Ringraziamenti

Ringrazio innanzitutto la mia famiglia che mi ha supportato moralmente ed economicamente, in particolar modo mio fratello Giacomo per l'aiuto datomi nella creazione di tutti i disegni.
Ringrazio la mia relatrice, la professoressa Cattabriga, per avermi aiutato nella stesura di questa tesi.
Ringrazio i miei amici per avermi accompagnato lungo tutto il percorso di studi.
Ringrazio infine la mia fidanzata Elisa, che mi ha sostenuto e che mi ha portato a scegliere ed approfondire questo argomento.