Modelli e algoritmi per il supporto delle decisioni

Un goliardico riassunto

Ollari Dmitri

25 maggio 2023

Indice

Capitolo 1

Introduzione ai modelli

1.1 Tipologie di algoritmi

1. Algoritmi costruttivi: Gli algoritmi costruttivi vengono utilizzati per costruire una soluzione in modo incrementale, aggiungendo passo dopo passo componenti o elementi al problema. Questi algoritmi partono da una soluzione vuota e aggiungono iterativamente elementi considerando regole o euristiche specifiche. Gli algoritmi costruttivi sono spesso efficienti dal punto di vista computazionale, ma potrebbero non garantire la soluzione ottimale. Tuttavia, possono essere utili quando è richiesta una soluzione veloce o approssimata.

2. Algoritmi di enumerazione: Gli algoritmi di enumerazione esplorano in modo sistematico tutte le possibili soluzioni di un problema, esaminando ogni possibile combinazione. Questi algoritmi possono essere implementati utilizzando tecniche come l'albero delle decisioni o la generazione di tutte le permutazioni. Gli algoritmi di enumerazione possono garantire la completezza (esaminando tutte le soluzioni) ma possono richiedere un tempo di esecuzione elevato per problemi di dimensioni significative.

3. Algoritmi di raffinamento locale: Gli algoritmi di raffinamento locale si concentrano sulla ricerca di soluzioni ottimali all'interno di un'area di ricerca limitata del problema. Questi algoritmi partono da una soluzione iniziale e iterativamente esplorano soluzioni vicine, apportando miglioramenti incrementali fino a quando non viene raggiunta una soluzione che soddisfa i criteri di ottimalità specificati. Gli algoritmi di raffinamento locale sono particolarmente efficaci per problemi in cui una piccola modifica nella soluzione può portare a un miglioramento sostanziale.

Capitolo 2

Introduzione ai grafi

Definizione: Un grafo è una struttura di dati utilizzata per rappresentare le relazioni tra oggetti. Formalmente, un grafo G è una coppia ordinata G = (V, E), dove V rappresenta l'insieme dei nodi (o vertici) e E rappresenta l'insieme degli archi. Gli archi possono essere diretti o non diretti, a seconda che rappresentino una connessione unidirezionale o bidirezionale tra i nodi.

Terminologia:

- **Grado di un nodo:** Il grado di un nodo è il numero di archi che sono connessi ad esso.
- **Percorso:** Un percorso è una sequenza di nodi collegati da archi.
- **Ciclo:** Un ciclo è un percorso che inizia e termina nello stesso nodo.
- **Grafo connesso:** Un grafo connesso è un grafo in cui esiste un percorso tra ogni coppia di nodi.
- **Grafo pesato:** Un grafo pesato è un grafo in cui ogni arco ha un peso o un valore associato.
- **Grafo orientato:** Un grafo orientato è un grafo in cui gli archi hanno una direzione specifica.

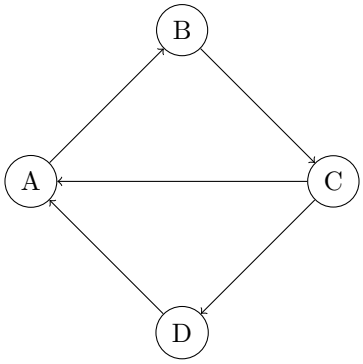
Rappresentazioni:

- **Matrice di adiacenza:** Una matrice bidimensionale che rappresenta la presenza o l'assenza di un arco tra i nodi.
- **Lista di adiacenza:** Una lista in cui ogni nodo ha un elenco dei suoi nodi adiacenti.
- **Lista degli archi:** Una lista che contiene tutte le coppie di nodi connessi da un arco.

La teoria dei grafi offre un ampio spettro di algoritmi e tecniche per lo studio e l'analisi dei grafi. Tra questi, ci sono algoritmi di attraversamento del grafo come la ricerca in profondità (DFS) e la ricerca in ampiezza (BFS), algoritmi per il percorso più breve come l'algoritmo di Dijkstra e algoritmi per la costruzione di alberi di copertura minimi come l'algoritmo di Kruskal.

La teoria dei grafi è una disciplina fondamentale con numerose applicazioni pratiche, che permette di risolvere una vasta gamma di problemi e modellare relazioni complesse tra gli oggetti.

2.1 Rappresentazione grafica



2.2 Liste di adiacenza

Le liste di adiacenza sono una delle rappresentazioni più comuni per i grafi. In questa rappresentazione, ogni nodo del grafo è associato a una lista di nodi adiacenti, cioè i nodi che sono direttamente collegati a quel nodo.

Per capire meglio, consideriamo un esempio di un grafo non diretto con 4 nodi (A, B, C, D) e 5 archi (AB, AC, AD, BC, CD). Ecco come potrebbe essere rappresentato utilizzando le liste di adiacenza:

```
A: [B, C, D]
B: [A, C]
C: [A, B, D]
D: [A, C]
```

Nell'esempio sopra, ogni nodo è seguito da una lista di nodi adiacenti. Ad esempio, il nodo A ha tre nodi adiacenti, che sono B, C e D.

La rappresentazione tramite liste di adiacenza presenta alcuni vantaggi. In primo luogo, è efficiente per i grafi sparsi, cioè quei grafi in cui il numero di archi è molto inferiore al numero massimo possibile di archi. Inoltre, consente di accedere rapidamente ai nodi adiacenti di un dato nodo. Tuttavia, può richiedere più spazio di memoria rispetto ad altre rappresentazioni come le matrici di adiacenza se il grafo ha molti archi.

Le liste di adiacenza sono utilizzate in numerosi algoritmi di grafi, come la ricerca in ampiezza (BFS) e la ricerca in profondità (DFS), poiché consentono un accesso efficiente ai vicini di un nodo.

2.3 Matrici di incidenza nodo arco

Le matrici di incidenza nodo-arco sono una rappresentazione comune per i grafi. In questa rappresentazione, le righe della matrice rappresentano i nodi del grafo e le colonne rappresentano gli archi. Gli elementi della matrice indicano l'incidenza dei nodi sugli archi, cioè se un nodo è collegato o meno a un particolare arco.

Per comprendere meglio, consideriamo un esempio di un grafo diretto con 4 nodi (A, B, C, D) e 5 archi (AB, AC, AD, BC, CD). Ecco come potrebbe essere rappresentato utilizzando una matrice di incidenza nodo-arco:

	AB	AC	AD	BC	CD
A	-1	0	0	0	0
B	1	-1	0	0	0
C	0	1	-1	0	0
D	0	0	1	-1	1

Nell'esempio sopra, ogni riga della matrice rappresenta un nodo, mentre ogni colonna rappresenta un arco. Gli elementi della matrice possono assumere i seguenti valori:

- -1: indica che il nodo è la sorgente dell'arco.
- 1: indica che il nodo è la destinazione dell'arco.
- 0: indica che il nodo non è collegato all'arco.

La rappresentazione tramite matrici di incidenza nodo-arco è particolarmente utile per i grafi con archi diretti e pesati. Questa rappresentazione consente di visualizzare facilmente le connessioni tra i nodi e gli archi del grafo. Tuttavia, rispetto alle liste di adiacenza o alle matrici di adiacenza, le matrici di incidenza richiedono più spazio di memoria e possono essere meno efficienti per alcune operazioni, come la ricerca dei vicini di un nodo.

Le matrici di incidenza nodo-arco sono utilizzate in diversi algoritmi e applicazioni dei grafi, come la risoluzione dei problemi di flusso massimo/minimo e la modellazione di reti di trasporto e comunicazione.

2.4 Archi adiacenti e cammini

Gli archi adiacenti si riferiscono agli archi che condividono un nodo in comune. In altre parole, due archi sono adiacenti se hanno un'estremità comune, che può essere un nodo di partenza o un nodo di arrivo. Ad esempio, se abbiamo un grafo con archi AB, AC e BC, gli archi AB e AC sono adiacenti poiché entrambi hanno il nodo A come estremità iniziale.

I cammini, d'altra parte, sono sequenze ordinate di nodi collegati da archi. Un cammino in un grafo è una serie di nodi in cui ogni coppia consecutiva di nodi è collegata da un arco. Possiamo distinguere diversi tipi di cammini:

- Cammino semplice: un cammino in cui nessun nodo appare più di una volta.

- Cammino elementare: un cammino in cui nessun arco appare più di una volta.
- Cammino ciclo: un cammino in cui il nodo di partenza coincide con il nodo di arrivo.

Ad esempio, considera un grafo con i nodi A, B, C e D e gli archi AB, BC e CD. Un possibile cammino in questo grafo potrebbe essere AB-BC-CD, che rappresenta un cammino semplice che connette il nodo A al nodo D attraverso i nodi B e C.

2.5 Circuiti hamiltoniani

Un circuito hamiltoniano in un grafo è un cammino chiuso che attraversa ogni nodo esattamente una volta, ad eccezione del nodo di partenza/arrivo che viene visitato due volte. In altre parole, è un percorso che visita tutti i nodi del grafo una sola volta, ritornando infine al nodo di partenza.

2.6 Componenti connesse

Le componenti connesse sono gruppi di nodi in un grafo che sono collegati tra loro attraverso archi. In altre parole, una componente connessa è un sottoinsieme di nodi di un grafo in cui esiste un percorso tra ogni coppia di nodi all'interno di quella componente.

2.6.1 Trovare le componenti connesse

```
def find_components(graph):
    visited = set()
    components = []

    def dfs(node, component):
        visited.add(node)
        component.append(node)

        for neighbor in graph[node]:
            if neighbor not in visited:
                dfs(neighbor, component)

    for node in graph:
        if node not in visited:
            current_component = []
            dfs(node, current_component)
            components.append(current_component)

    return components

graph = {
    'A': ['B', 'C'],
    'B': ['A', 'C'],
    'C': ['A', 'B'],
    'D': ['E'],
    'E': ['D']
}

components = find_components(graph)
print(components)
```

2.7 Grafo completo

Un grafo completo è un tipo particolare di grafo non diretto in cui ogni coppia di nodi è collegata da un arco. In altre parole, in un grafo completo, ogni nodo è connesso direttamente a tutti gli altri nodi del grafo.

2.8 Matching

In teoria dei grafi, un *matching* in un grafo $G = (V, E)$ è un insieme di archi $M \subseteq E$ tale che nessun nodo condivide gli estremi di due archi in M.

Formalmente, dato un grafo non diretto $G = (V, E)$, un matching M in G soddisfa le seguenti condizioni:

1. Per ogni arco $(u, v) \in M$, nessun arco in M condivide il nodo iniziale u o il nodo finale v.
2. Ogni nodo in V è incidente a al più un arco in M.

Un *matching massimale* è un matching che non può essere esteso aggiungendo ulteriori archi senza violare la proprietà di non sovrapposizione. Un *matching perfetto* è un matching in cui tutti i nodi del grafo sono coperti da un arco del matching.

Il matching ha diverse applicazioni, come problemi di assegnazione, progettazione di reti, teoria dei giochi e altro ancora.

2.9 Grafi bipartiti

Un grafo bipartito è un tipo di grafo non diretto in cui i nodi possono essere divisi in due insiemi disgiunti, in modo che ogni arco del grafo colleghi un nodo di un insieme all'altro.

Formalmente, un grafo bipartito $G = (V, E)$ è definito da due insiemi di nodi disgiunti $V1$ e $V2$, dove gli archi del grafo collegano solo i nodi di $V1$ con i nodi di $V2$.

I grafi bipartiti sono rappresentati graficamente come insiemi di punti su due righe parallele, dove gli archi collegano i punti delle due righe.

I grafi bipartiti hanno diverse proprietà interessanti, come la possibilità di essere colorati con solo due colori e l'assenza di cicli di lunghezza dispari.

2.9.1 Trovare grafi bipartiti

```

def is_bipartite(graph):
    color = {}
    visited = set()

    def dfs(node, c):
        visited.add(node)
        color[node] = c

        for neighbor in graph[node]:
            if neighbor not in visited:
                if not dfs(neighbor, 1 - c):
                    return False
            elif color[neighbor] == color[node]:
                return False

        return True

    for node in graph:
        if node not in visited:
            if not dfs(node, 0):
                return False

    return True


graph = {
    'A': ['B', 'C'],
    'B': ['A', 'D'],
    'C': ['A', 'D'],
    'D': ['B', 'C']
}

```

```
34 if is_bipartite(graph):
35   print("bipartito")
36 else:
37   print("non bipartito")
```

2.10 Alberi

Un albero è una struttura dati gerarchica che consiste in un insieme di nodi collegati tra loro in modo specifico. L'albero è composto da una radice, nodi, e archi che connettono i nodi.

I principali termini associati agli alberi sono:

- **Radice**: il nodo di partenza dell'albero.
- **Nodi**: gli elementi dell'albero.
- **Archi**: i collegamenti tra i nodi dell'albero.
- **Figli**: i nodi direttamente collegati a un dato nodo.
- **Genitore**: il nodo da cui un dato nodo è raggiungibile tramite un arco.
- **Foglie**: i nodi che non hanno figli.
- **Livello**: la distanza tra un nodo e la radice.
- **Altezza**: il numero massimo di livelli dell'albero.

Gli alberi sono utilizzati in diverse applicazioni, come la rappresentazione delle directory nei sistemi operativi, la gerarchia delle organizzazioni aziendali e la struttura dei dati nelle basi di dati.

2.11 Alberi di supporto

Gli alberi di supporto, noti anche come alberi di copertura o alberi di connessione minima, sono alberi speciali utilizzati in teoria dei grafi. Sono utilizzati per connettere tutti i nodi di un grafo con un numero minimo di archi, garantendo al contempo la connessione tra tutti i nodi.

Formalmente, dato un grafo non diretto ponderato $G = (V, E)$, dove V rappresenta l'insieme dei nodi e E rappresenta l'insieme degli archi, un albero di supporto di G è un sottografo che soddisfa le seguenti proprietà:

1. L'albero di supporto contiene tutti i nodi di G.
2. L'albero di supporto è un albero, cioè non contiene cicli.
3. L'albero di supporto ha il minor peso totale tra tutti gli alberi che soddisfano le prime due proprietà.

Gli alberi di supporto sono utili in diversi contesti, come reti di comunicazione, logistica, progettazione di reti stradali e molto altro ancora.

Capitolo 3

Minimum Spanning Tree

Un Minimum Spanning Tree (MST), tradotto in italiano come Albero di Supporto Minimo, è un albero di supporto di peso minimo in un grafo non diretto e connesso. L'MST è costituito da un sottoinsieme di archi del grafo che connette tutti i nodi in modo tale che il peso totale degli archi sia il più basso possibile.

Le caratteristiche principali degli MST sono:

1. **Connessione**: Un MST deve connettere tutti i nodi del grafo, garantendo che non ci siano nodi isolati.
2. **Aciclicità**: Un MST non deve contenere cicli, quindi non può avere archi che creano loop all'interno dell'albero.
3. **Peso minimo**: Un MST ha il peso totale degli archi più basso possibile tra tutti gli alberi che soddisfano le prime due caratteristiche.

Gli MST hanno diverse applicazioni pratiche, tra cui:

- Reti di comunicazione: Un MST può essere utilizzato per collegare un insieme di punti in una rete di comunicazione minimizzando il costo totale dei collegamenti.
- Progettazione di reti stradali: Gli MST possono essere utilizzati per pianificare reti stradali efficienti, dove gli archi rappresentano le strade e il peso degli archi può essere la distanza o il tempo di percorrenza.
- Analisi dei dati: Gli MST possono essere utilizzati per individuare le relazioni più rilevanti o significative tra i dati, ad esempio nella visualizzazione delle relazioni tra punti di dati su una mappa.

3.1 Algoritmo greedy per MST

L'algoritmo greedy per MST (Minimum Spanning Tree) è un approccio basato sulla selezione di archi in base al loro peso. L'idea principale è quella di selezionare ripetutamente l'arco di peso minimo che collega un nodo dell'MST esistente a un nodo non ancora raggiunto, finché non viene creato un albero che connette tutti i nodi del grafo.

1. Inizializzazione: Parti da un grafo non diretto e connesso $G = (V, E)$, dove V rappresenta l'insieme dei nodi e E rappresenta l'insieme degli archi. Crea un insieme vuoto MST che conterrà gli archi dell'albero di supporto minimo.
2. Seleziona un nodo di partenza arbitrario. Questo può essere fatto in modo casuale o seguendo una strategia specifica, ad esempio selezionando il primo nodo dell'insieme dei nodi.
3. Finché non sono stati raggiunti tutti i nodi:
   (a) Seleziona l'arco di peso minimo che collega un nodo nell'MST esistente a un nodo non ancora raggiunto. Questo arco deve essere selezionato tra gli archi che collegano il nodo raggiunto all'esterno dell'MST.
   (b) Aggiungi l'arco selezionato all'MST.
   (c) Marca il nodo raggiunto come "visitato" o "raggiunto".
4. Alla fine del processo, l'MST conterrà tutti gli archi necessari per connettere tutti i nodi del grafo in modo che il peso totale dell'MST sia minimo.

L'algoritmo greedy per MST può essere implementato utilizzando diverse strutture dati, come ad esempio una coda di priorità (heap) per selezionare l'arco di peso minimo in modo efficiente.

Ecco un esempio di implementazione dell'algoritmo greedy per MST in Python:

```
1
2  def greedy_mst(graph):
3      mst = []  # Insieme di archi dell'MST
4      visited = set()  # Insieme di nodi visitati
5      start_node = list(graph.keys())[0]  # Nodo di partenza arbitrario
6
7      visited.add(start_node)
8
9      while len(visited) < len(graph):
10         min_weight = float('inf')
11         min_edge = None
12
13         # Scansiona gli archi collegati ai nodi visitati
14         for node in visited:
15             for neighbor, weight in graph[node]:
16                 if neighbor not in visited and weight < min_weight:
17                     min_weight = weight
18                     min_edge = (node, neighbor)
19
20         if min_edge:
21             mst.append(min_edge)
22             visited.add(min_edge[1])
23
24     return mst
```

Questo è solo un esempio di implementazione e può variare a seconda delle specifiche del problema. Assicurati di adattare l'algoritmo in base alle tue esigenze specifiche.

3.1.1 Correttezza algoritmo

La complessità dell'algoritmo greedy per MST dipende dalla rappresentazione del grafo e dalla struttura dati utilizzata.

Assumendo che il grafo sia rappresentato come una lista di adiacenza, con n nodi e m archi, e che si utilizzi una coda di priorità (heap) per selezionare l'arco di peso minimo, la complessità dell'algoritmo è:

- Inizializzazione: $O(n)$
- Ciclo principale: $O(m \log m)$

All'interno del ciclo principale, l'estrazione dell'arco di peso minimo richiede un'operazione di estrazione minima dalla coda di priorità, che ha una complessità di $O(\log m)$. Il ciclo principale viene eseguito m volte.

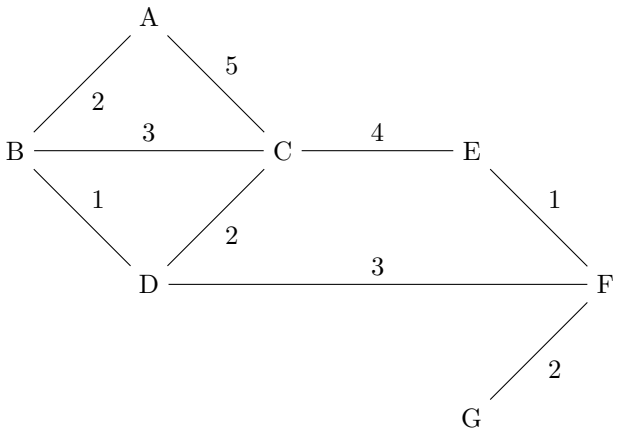
Quindi, la complessità complessiva dell'algoritmo greedy per MST è $O(n + m \log m)$.

È importante notare che se il grafo viene rappresentato come una matrice di adiacenza invece di una lista di adiacenza, la complessità può essere diversa, ad esempio $O(n^2)$ per la ricerca dell'arco di peso minimo.

3.1.2 Complessitá dell'algoritmo

3.2 Algoritmo MST-1 - Prim

(Penso sia l'algoritmo di Prim ma al prof piace inventarsi della roba a caso)


A
5
2
3
B
C
4
E
1
1
2
3
D
F
2
G