Modelli e algoritmi per il supporto delle decisioni

Un goliardico riassunto

Ollari Dmitri

30 maggio 2023

Indice

1 Introduzione ai modelli **3**
1.1 Tipologie di algoritmi . . . 3

2 Introduzione ai grafi **4**
2.1 Rappresentazione grafica . . . 4
2.2 Liste di adiacenza . . . 5
2.3 Matrici di incidenza nodo arco . . . 5
2.4 Archi adiacenti e cammini . . . 5
2.5 Circuiti hamiltoniani . . . 6
2.6 Componenti connesse . . . 6
2.6.1 Trovare le componenti connesse . . . 6
2.7 Grafo completo . . . 6
2.8 Matching . . . 7
2.9 Grafi bipartiti . . . 7
2.9.1 Trovare grafi bipartiti . . . 7
2.10 Alberi . . . 8
2.11 Alberi di supporto . . . 8

3 Minimum Spanning Tree **9**
3.1 Algoritmo greedy per MST . . . 9
3.1.1 Correttezza algoritmo . . . 10
3.1.2 Complessitá dell'algoritmo . . . 11
3.2 Algoritmo MST-1 - Prim . . . 11
3.2.1 Procedimento . . . 11
3.3 Algoritmo MST-2 - Kruskal . . . 11
3.3.1 Procedimento . . . 11

4 Algoritmi di cammini minimi **12**
4.1 Algoritmo di Dijkstra . . . 12
4.1.1 Complessità . . . 12
4.2 Algoritmo di Floyd-Warshall . . . 12

5 Matching **13**

6 Assegnamento **14**
6.1 Problema . . . 14
6.2 Matrice dei costi . . . 14
6.2.1 Riduzione della matrice dei costi . . . 14
6.3 Algoritmo ungherese . . . 15

7 Branch and Bound **16**
7.1 Compontenti dell'algoritmo . . . 16
7.1.1 Upper Bound . . . 16
7.1.2 Lower Bound . . . 16
7.1.3 Valore Obiettivo . . . 16
7.2 Algoritmo . . . 17

8 Problema KNAPSACK **18**
8.1 Brach and bound per KNAPSACK . . . 18

9 Programmazione dinamica **20**
9.1 Il principio di ottimalità 20
9.2 Programmazione dinamica per il problema dello zaino 20
9.3 Che cos'è l'algoritmo valore ottimo? 21
9.3.1 Applicato a KNAPSACK 21
9.4 Problema della schedulazione 21

10 Problemi di ottimizzazione **23**
10.1 La classe P 23
10.2 La classe NP 23
10.3 La classe NP-completi 23

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.

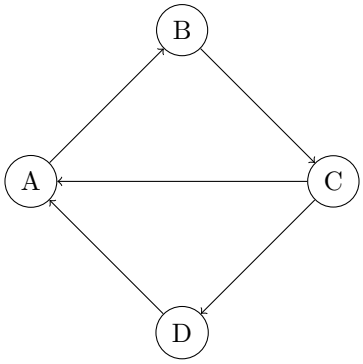
Rappresentazioni:

- **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: $O(m \log m)$

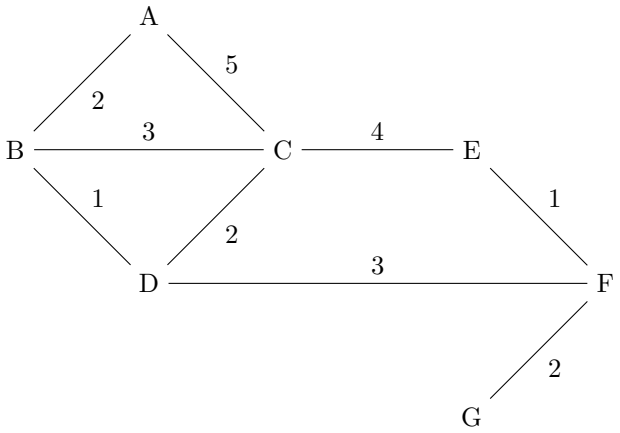
All'interno del ciclo principale, l'estrazione dell'arco di peso minimo richiede un'operazione di estrazione minima dalla coda di priorità, che ha una complessità di $O(\log m)$. Il ciclo principale viene eseguito m volte.

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

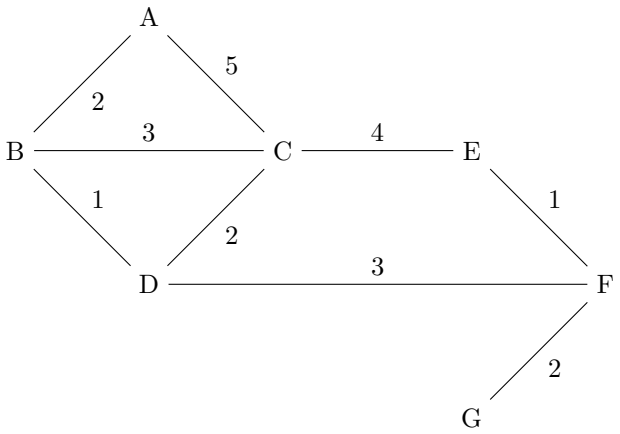
3.2 Algoritmo MST-1 - Prim



3.2.1 Procedimento

1. Sceglie un nodo qualsiasi come nodo di partenza e lo aggiunge alla lista dei nodi visitati
2. Scegliere un nodo non visitato che abbia un arco che lo colleghi a un nodo visitato con peso minimo e aggiungerlo alla lista dei nodi visitati
3. Ripetere il passo 2 fino a quando non sono stati visitati tutti i nodi

3.3 Algoritmo MST-2 - Kruskal



3.3.1 Procedimento

1. Si ordinano tutti gli archi del grafo in ordine crescente del peso.
2. Si aggiunge un arco alla volta alla lista degli archi dell'albero di supporto, a patto che non si creino cicli.
3. Si ripete il passo 2 fino a che non si sono aggiunti $n - 1$ archi, dove n è il numero di nodi del grafo.

Capitolo 4

Algoritmi di cammini minimi

Ci sono due algotimi principali per risolvere il problema del cammino minimo:

- Algoritmo di Dijkstra: valido per grafi con pesi non negativi
- Algoritmo di Floyd-Warshall: valido per grafi con anche pesi negativi

4.1 Algoritmo di Dijkstra

Percorso minimo da un nodo s a tutti i nodi del grafo(grafo con pesi non negativi).

1. Si sceglie un nodo di partenza s e si inizializza un vettore d con la distanza di tutti i nodi da s.
2. Si sceglie il nodo v con distanza minima da s e si aggiorna il vettore d con le distanze dei nodi adiacenti a v.
3. Si ripete il passo 2 fino a che non sono stati visitati tutti i nodi.

4.1.1 Complessità

L'algoritmo di Dijkstra ha complessità $O(n^2)$ se il grafo è rappresentato con una matrice di adiacenza, $O(m \log n)$ se il grafo è rappresentato con una lista di adiacenza.

4.2 Algoritmo di Floyd-Warshall

Percorso minimo tra tutti i nodi del grafo(grafo con pesi anche negativi).
DA RIFARE

Capitolo 5

Matching

Capitolo 6

Assegnamento

6.1 Problema

L'assegnamento è un problema di ottimizzazione che consiste nel trovare la soluzione ottima di un insieme di assegnamenti tra un gruppo di lavoratori e un gruppo di mansioni, considerando i costi associati ad ogni possibile assegnamento. In altre parole, l'assegnamento cerca di minimizzare il costo totale dell'assegnamento dei lavoratori alle mansioni, soddisfacendo allo stesso tempo i vincoli di assegnamento.

6.2 Matrice dei costi

Per questo tipo di task si utilizzano delle matrici che rappresentano i costi,

	b_1	b_2	b_3	b_4	b_5
a_1	c_{11}	c_{12}	c_{13}	c_{14}	c_{15}
a_2	c_{21}	c_{22}	c_{23}	c_{24}	c_{25}
a_3	c_{31}	c_{32}	c_{33}	c_{34}	c_{35}
a_4	c_{41}	c_{42}	c_{43}	c_{44}	c_{45}

6.2.1 Riduzione della matrice dei costi

Per effettuare la riduzione della matrice dei costi occorre selezionare l'elemento con il costo minore all'interno dell'intera matrice e sottastrarlo a tutti gli altri elementi della matrice.

	b_1	b_2	b_3
a_1	5	2	4
a_2	2	4	3
a_3	4	3	6

Dove l'elemento con il costo minore è 2 e quindi la matrice diventa:

	b_1	b_2	b_3
a_1	5	2	4
a_2	2	4	3
a_3	4	3	6
d_1	2	2	3

Il risulato di questa operazione è una matrice ridotta, che contiene almeno uno zero per ogni riga e colonna.

	b_1	b_2	b_3
a_1	3	0	1
a_2	0	2	0
a_3	2	1	3

Questa operazione di riduzione si può applicare anche aggiungendo una colonna.

6.3 Algoritmo ungherese

L'algoritmo ungherese, noto anche come algoritmo di assegnazione o algoritmo Munkres, è un algoritmo utilizzato per risolvere il problema dell'assegnazione ottima in un problema di assegnazione. Questo problema riguarda l'assegnazione di un insieme di elementi a un altro insieme di elementi, minimizzando o massimizzando una funzione di costo associata.

Algoritmo Ungherese (Algoritmo di Assegnazione)
Input: Una matrice dei costi di dimensioni $N \times N$
Output: Un assegnamento ottimo
Inizializza una matrice ausiliaria dei costi con gli stessi valori della matrice dei costi;
while *Non è stata ottenuta una soluzione ottima* **do**
  Fase 1: Riduzione dei costi;
  for *Ogni riga i della matrice dei costi* **do**
    Sottrai il valore minimo della riga i a tutti gli elementi della riga;
  end
  for *Ogni colonna j della matrice dei costi* **do**
    Sottrai il valore minimo della colonna j a tutti gli elementi della colonna;
  end
  Fase 2: Copertura delle righe e assegnazione;
  Copri le righe e assegna le celle seguendo le regole:;
  while *Esiste una cella non coperta* **do**
    Trova una cella non coperta C con il costo minimo nella sua riga e colonna;
    if *La riga di C non è coperta* **then**
      Copri la riga di C;
    end
    Assegna la cella C;
    while *Esiste una riga coperta che contiene una cella assegnata* **do**
      Trova una cella assegnata nella stessa riga;
      Assegna la cella trovata;
    end
  end
  Fase 3: Aggiornamento dei costi;
  if *Il numero di assegnazioni effettuate è minore di N* **then**
    Calcola il valore minimo tra gli elementi non coperti della matrice dei costi;
    Sottrai il valore minimo a tutti gli elementi non coperti;
    Aggiungi il valore minimo a tutti gli elementi coperti da due linee (riga e colonna);
  end
end
return *L'assegnamento ottenuto*;

Algorithm 1: Algoritmo Ungherese (Algoritmo di Assegnazione)

Questo algoritmo risolve il problema dell'assegnazione ottima, riducendo i costi e assegnando le celle in base a determinate regole. Le fasi 1 e 2 vengono eseguite finché non viene ottenuta una soluzione ottima, e la fase 3 viene eseguita solo se necessario. Alla fine, l'algoritmo restituisce un assegnamento ottimo.

Capitolo 7

Branch and Bound

Questa tecnica si utilizza quando si devono risolvere problemi di programmazione lineare intera (PLI) o di programmazione mista intera (PMI). Il suo funzionamento si basa su una strategia di divide et impera, che consiste nel suddividere il problema in sotto-problemi più piccoli e risolvibili più facilmente.

7.1 Compontenti dell'algoritmo

7.1.1 Upper Bound

L'upper bound rappresenta un limite superiore per il valore della soluzione ottima. Viene utilizzato per stabilire un valore di riferimento iniziale per valutare le soluzioni parziali e confrontarle con la soluzione ottima corrente. Durante l'esecuzione dell'algoritmo, se viene trovata una soluzione parziale che supera il limite superiore corrente, il sottoproblema associato a quella soluzione può essere scartato, poiché non può migliorare la soluzione ottima trovata finora. L'obiettivo è ridurre il limite superiore durante l'esecuzione dell'algoritmo, fino a raggiungere il valore della soluzione ottima.

7.1.2 Lower Bound

Il lower bound rappresenta un limite inferiore per il valore della soluzione ottima. Viene calcolato per ogni sottoproblema generato durante l'esecuzione dell'algoritmo. Il lower bound può essere ottenuto utilizzando euristiche o algoritmi di approssimazione per stimare il valore minimo che può essere ottenuto dalla soluzione ottima in quel sottoproblema. Se il lower bound di un sottoproblema è maggiore del limite superiore corrente, il sottoproblema e il suo sottoalbero di soluzioni possono essere eliminati dall'esplorazione, poiché non possono migliorare la soluzione ottima trovata finora.

7.1.3 Valore Obiettivo

Il valore obiettivo è il valore associato alla soluzione ottima del problema di ottimizzazione. L'obiettivo dell'algoritmo Branch and Bound è trovare una soluzione che abbia il valore obiettivo migliore possibile. Durante l'esecuzione dell'algoritmo, se viene trovata una soluzione parziale con un valore obiettivo migliore del limite superiore corrente, il limite superiore viene aggiornato con il nuovo valore obiettivo. L'obiettivo finale è raggiungere la soluzione ottima con il valore obiettivo massimo o minimo, a seconda del tipo di problema di ottimizzazione.

7.2 Algoritmo

Input: Problema di ottimizzazione
Output: Soluzione ottima
Inizializza una soluzione parziale vuota; Inizializza un limite superiore iniziale ad un valore molto alto;
while *Non sono stati esplorati tutti i sottoproblemi* **do**
    Genera un nuovo sottoproblema più piccolo; Calcola un limite inferiore per il sottoproblema;
    if *Limite inferiore* > *Limite superiore* **then**
        Non esplorare ulteriormente il sottoproblema;
    end
    else
        Esplora il sottoproblema generando ulteriori soluzioni parziali; **if** *Soluzione parziale completa* **then**
            if *Valore obiettivo* < *Limite superiore* **then**
                Aggiorna il limite superiore con il valore obiettivo;
            end
        end
    end
end
return *Soluzione ottima corrispondente al limite superiore migliore*;

Algorithm 2: Algoritmo Branch and Bound

Capitolo 8

Problema KNAPSACK

Il problema dello zaino (in inglese, knapsack problem) è un problema di ottimizzazione combinatoria. Si supponga di avere uno zaino con una capacità massima e una serie di oggetti, ognuno dei quali ha un determinato valore e un certo peso. Lo scopo è riempire lo zaino con gli oggetti in modo da massimizzare il valore totale degli oggetti, rispettando la capacità massima dello zaino.

Formalmente, il problema può essere definito come segue: dati n oggetti con un valore v_i e un peso w_i per $i = 1, ..., n$ e una capacità massima W dello zaino, trovare una combinazione di oggetti che massimizzi il valore totale degli oggetti, rispettando la capacità massima dello zaino. In altre parole, si cerca di trovare una soluzione x, dove $x_i \in \{0, 1\}$ rappresenta se l'oggetto i è incluso o meno nello zaino, tale che

$$\sum_{i=1}^{n} w_i x_i \leq W$$

$$\max \sum_{i=1}^{n} v_i x_i$$

dove $\sum_{i=1}^{n} w_i x_i \leq W$ rappresenta il vincolo sulla capacità dello zaino.

Il problema dello zaino è un problema di ottimizzazione combinatoria NP-completo, il che significa che non esiste un algoritmo che possa risolvere il problema in un tempo polinomiale in base alla dimensione dell'input. Tuttavia, esistono algoritmi efficienti per risolvere il problema in alcuni casi particolari, come quando tutti i pesi e i valori degli oggetti sono interi positivi.

8.1 Brach and bound per KNAPSACK

L'algoritmo di branch and bound applicato al problema dello zaino (knapsack problem) prevede i seguenti passi:

1. Inizializzazione: si parte dalla soluzione vuota e si calcola l'upper bound (UB) iniziale come la soluzione ottenuta con il rilassamento continuo del problema.

2. Branching: si seleziona un elemento non ancora selezionato e si generano due sotto-problemi, uno in cui l'elemento viene inserito nello zaino e uno in cui non viene inserito. Per ogni sotto-problema, si calcola l'upper bound e si scarta il sotto-problema se l'upper bound è minore della migliore soluzione trovata finora.

3. Selezione del sotto-problema: si seleziona il sotto-problema con l'upper bound più alto tra quelli non ancora scartati e si ripete il processo dal passo 2.

4. Terminazione: l'algoritmo termina quando tutti i sotto-problemi sono stati esplorati o quando non ci sono più sotto-problemi con un upper bound maggiore della migliore soluzione trovata finora.

Durante l'esplorazione dell'albero dei sotto-problemi, l'upper bound viene calcolato come la somma dei valori degli elementi già selezionati e del valore massimo che si può ancora ottenere con gli elementi rimanenti, tenendo conto del limite di peso dello zaino.

La soluzione ottimale viene ottenuta quando l'algoritmo termina e restituisce la migliore soluzione trovata finora.

Input: Array di n oggetti con peso p_i e valore v_i, capacità dello zaino W
Output: Valore massimo che si può ottenere riempiendo lo zaino
inizializzazione: $u \leftarrow 0, LB \leftarrow 0, x \leftarrow \vec{0}, j \leftarrow 1, z \leftarrow 0$
Heap : tutti i nodi sono presenti nel heap con LB come chiave
while *Heap non vuoto* **do**
    $N \leftarrow$ nodo con il valore minimo di LB
    if $LB \geq u$ **then**
        return u
    end
    $i \leftarrow$ livello di profondità del nodo N
    if $i > n$ **then**
        continue
    end
    if $z + v_i \leq u$ **then**
        continue
    end
    crea il figlio sinistro S del nodo N mettendo nell'array x il valore 0 per l'oggetto i
    crea il figlio destro D del nodo N mettendo nell'array x il valore 1 per l'oggetto i
    calcola il lower bound LB_S del figlio sinistro S
    calcola il lower bound LB_D del figlio destro D
    if $LB_S < u$ **then**
        inserisci S nel heap con LB_S come chiave
    end
    if $LB_D < u$ **then**
        inserisci D nel heap con LB_D come chiave
    end
end
return u

Algorithm 3: Algoritmo Branch and Bound per il problema dello zaino (Knapsack)

Capitolo 9

Programmazione dinamica

La programmazione dinamica è applicabile a problemi che rispettano:

- Il problema può essere suddiviso in sottoproblemi più piccoli.
- In ogni sotto blocco k, con $k = 1, \ldots, n$, ci si trova in uno degli stati possibili S_k.
- In ogni blocco si deve prendere una decisione d_k che appartiene al dominio delle decisioni D_k.

In ogni blocco k si ha che la funzione obiettivo f_k é $u(d_k, s_k)$
Se in un momento mi trovo nel blocco k con decisione d_k e stato s_k, posso passare allo stato successivo $s_{k+1} = t(d_k, s_k)$ dove la funzione di transizione t é definita come funzione di transizione.

9.1 Il principio di ottimalità

Il principio di ottimalità è un concetto fondamentale della programmazione dinamica. Esso afferma che una soluzione ottima a un problema di ottimizzazione globale può essere costruita attraverso le soluzioni ottime dei suoi sotto-problemi. In altre parole, se un problema può essere suddiviso in sotto-problemi più piccoli, la soluzione ottima del problema globale può essere ottenuta combinando le soluzioni ottime dei suoi sotto-problemi.

Questo principio si applica quando si hanno problemi in cui la soluzione ottima di una istanza del problema contiene al suo interno la soluzione ottima di sotto-istanze del problema stesso. In questo caso, la soluzione del problema può essere ottenuta risolvendo le sotto-istanze e combinando le loro soluzioni in modo opportuno.

L'utilizzo del principio di ottimalità permette di evitare di risolvere più volte lo stesso sotto-problema, riducendo così il tempo di calcolo e aumentando l'efficienza dell'algoritmo di programmazione dinamica.

9.2 Programmazione dinamica per il problema dello zaino

Si consideri il problema dello zaino in cui si ha a disposizione uno zaino di capacità C e un insieme di n oggetti. Ogni oggetto i ha un peso p_i e un valore v_i. Si vuole trovare la combinazione di oggetti che massimizza il valore totale, rispettando la capacità dello zaino.

Definiamo $K(i, w)$ la soluzione ottima del problema dello zaino utilizzando i primi i oggetti e uno zaino di capacità w. Il problema può essere risolto tramite programmazione dinamica utilizzando il principio di ottimalità.

Il principio di ottimalità afferma che una soluzione ottima al problema dello zaino che considera i primi i oggetti, è ottenuta considerando o l'oggetto i o meno. Quindi, la soluzione ottima può essere ottenuta confrontando il valore massimo che si può ottenere considerando l'oggetto i (e utilizzando uno zaino di capacità $w - p_i$) con il valore massimo che si può ottenere senza considerare l'oggetto i (e utilizzando uno zaino di capacità w). Formalmente:

$$K(i, w) = \begin{cases} 0 & i = 0 \text{ o } w = 0 \\ K(i-1, w) & p_i > w \\ \max\{K(i-1, w), K(i-1, w-p_i) + v_i\} & \text{altrimenti} \end{cases}$$

La soluzione al problema dello zaino può essere trovata calcolando $K(n, C)$.

Inoltre, è possibile utilizzare la programmazione dinamica per trovare la combinazione di oggetti che massimizza il valore totale. Dopo aver calcolato la matrice $K(i, w)$, si può risalire ai singoli oggetti utilizzati nella soluzione ottima tramite la seguente procedura:

Result: Oggetti utilizzati nella soluzione ottima
$i \leftarrow n$;
$w \leftarrow C$;
while $i > 0$ **do**
  if $K(i, w) \neq K(i-1, w)$ **then**
    Utilizza l'oggetto i;
    $w \leftarrow w - p_i$;
  end
  $i \leftarrow i - 1$;
end

Algorithm 4: Procedura per ottenere gli oggetti utilizzati nella soluzione ottima

9.3 Che cos'è l'algoritmo valore ottimo?

L'algoritmo valore ottimo (in inglese *Value Iteration*) è un algoritmo di programmazione dinamica utilizzato per trovare la politica ottima in un processo decisionale di Markov a tempo discreto (MDP). L'algoritmo è basato sull'iterazione dei valori e utilizza una procedura di backup dei valori per aggiornare il valore di ogni stato.

L'idea alla base dell'algoritmo valore ottimo è quella di trovare la funzione valore ottimo $V^*(s)$ di ogni stato s del MDP. Questa funzione indica il valore atteso della ricompensa totale che si può ottenere partendo dallo stato s e seguendo la politica ottima. L'algoritmo itera la stima della funzione valore ottimo fino a raggiungere una convergenza. Alla fine di ogni iterazione, l'algoritmo aggiorna la funzione valore ottimo di ogni stato sulla base del valore atteso dei suoi successori.

L'algoritmo valore ottimo è un metodo molto efficace per risolvere MDPs di grandi dimensioni, tuttavia richiede la conoscenza completa del modello del sistema, ovvero delle probabilità di transizione e delle ricompense associate ad ogni transizione.

9.3.1 Applicato a KNAPSACK

Input: Un insieme di n oggetti, dove ogni oggetto i ha un valore v_i e un peso w_i, e una capacità massima dello zaino W
Output: Il valore massimo che può essere ottenuto con una combinazione di oggetti che non superi la capacità W
$A \leftarrow$ array $n \times (W+1)$ inizializzato a 0;
for $i \leftarrow 1$ **to** n **do**
  for $w \leftarrow 1$ **to** W **do**
    if $w_i \leq w$ **then**
      $A[i, w] \leftarrow \max\{A[i-1, w], A[i-1, w-w_i] + v_i\}$;
    else
      $A[i, w] \leftarrow A[i-1, w]$;
    end
  end
end
return $A[n, W]$

Algorithm 5: Algoritmo valore ottimo per il problema dello zaino

In questo algoritmo, si utilizza una matrice A di dimensioni $n \times (W+1)$ per memorizzare i valori ottimi per tutti i sottoproblemi del problema dello zaino, ovvero il valore massimo che può essere ottenuto con una combinazione di oggetti che non superi una capacità $w \leq W$ e che includa solo i primi i oggetti.

L'algoritmo utilizza una doppia iterazione sui primi i oggetti e sulle capacità w. Se l'oggetto i ha peso w_i minore o uguale alla capacità w considerata, allora è possibile scegliere se includere o meno l'oggetto i nella combinazione. Se si decide di includerlo, il valore massimo ottenibile è dato dalla somma del valore dell'oggetto i e del valore massimo ottenibile con i primi $i-1$ oggetti e una capacità residua di $w - w_i$. Altrimenti, il valore massimo ottenibile è dato dal valore massimo ottenibile con i primi $i-1$ oggetti e la capacità w considerata.

Alla fine delle iterazioni, il valore ottimo per il problema originale, ovvero la combinazione di oggetti con valore massimo che non superi la capacità W, è dato dall'elemento $A[n, W]$ della matrice.

9.4 Problema della schedulazione

Il **problema della schedulazione** consiste nell'assegnare un insieme di n attività da eseguire, ognuna delle quali richiede un certo tempo di elaborazione, a un insieme di m risorse, ciascuna delle quali ha una certa disponibilità. L'obiettivo è minimizzare il tempo totale di completamento delle attività.

Formalmente, il problema può essere definito come segue:
Dati:

- Un insieme di n attività $A = \{a_1, a_2, ..., a_n\}$.
- Un insieme di m risorse $R = \{r_1, r_2, ..., r_m\}$.
- Un tempo di elaborazione p_{ij} per eseguire l'attività a_i sulla risorsa r_j.
- Una disponibilità d_j per ogni risorsa r_j.

Obiettivo: Trovare un assegnamento delle attività alle risorse che minimizzi il tempo totale di completamento.

Il problema della schedulazione può essere formulato in diverse varianti, come ad esempio il problema della schedulazione su una singola macchina o il problema della schedulazione su più macchine. In generale, il problema è considerato *NP-hard*, il che significa che non esiste un algoritmo efficiente in grado di risolverlo in tempo polinomiale per tutti i casi.

Capitolo 10

Problemi di ottimizzazione

I problemi di ottimizzazione sono un tipo di problemi matematici in cui si cerca di trovare il valore massimo o minimo di una funzione, nota come funzione obiettivo, in presenza di una serie di vincoli. Questi vincoli possono essere di natura diversa, ad esempio possono essere delle equazioni o delle disuguaglianze che limitano l'insieme di soluzioni ammissibili per il problema.

Ogni istanza é rappresentata come:

$$f : S \to R \tag{10.1}$$

Dove S é l'insieme delle soluzioni ammissibili e R é l'insieme dei valori che la funzione obiettivo puó assumere.

10.1 La classe P

Dato un problema di ottimizzazione R, diciamo che questo appartiene alla classe P se esiste un algoritmo A di complessitá polinomiale che lo risolve.

Alcuni esempi di problemi in P sono:

- **SHORT PATH**: dato un grafo $G = (V, E)$, un nodo $s \in V$ e un nodo $t \in V$, trovare il cammino piú breve da s a t.
- **MST**: dato un grafo $G = (V, E)$, trovare un albero di copertura minimo.

10.2 La classe NP

La classe NP contiene tutti i problemi di ottimizzazione per i quali, nota la soluzione ottima, é possibile calcolarlo in tempo polinomiale.

Alcuni esempi di problemi in NP sono:

- **CLIQUE**: dato un grafo $G = (V, E)$ e un intero k, trovare un sottoinsieme di k nodi di V che formano un grafo completo.
- **TSP**: dato un grafo $G = (V, E)$, trovare un ciclo hamiltoniano di costo minimo.

10.3 La classe NP-completi

Un problema R é NP-completo se:

- $R \in NP$
- $\forall Q \in NP \exists$ riduzione polinomiale di Q in R

Esempi di problemi NP-completi sono:

- **CLIQUE**: dato un grafo $G = (V, E)$ e un intero k, trovare un sottoinsieme di k nodi di V che formano un grafo completo.
- **TSP**: dato un grafo $G = (V, E)$, trovare un ciclo hamiltoniano di costo minimo.
- **KNAPSACK**: dato un insieme di n oggetti, ognuno con un peso w_i e un valore v_i, e un intero W, trovare un sottoinsieme di oggetti che non superi il peso W e che massimizzi il valore totale.