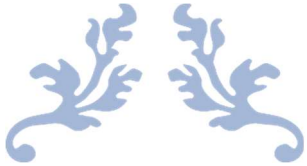


PROCESS SCHEDULING COME LONGEST PATH PROBLEM

Tesina di Graphs and Network Optimization

COCCO MATTIA 65336

2024/2025

Università degli studi di Cagliari

Introduzione e problema

Il problema trattato prende spunto dai calcolatori elettronici che impieghiamo ogni giorno, dove una serie di processi richiedono risorse computazionali e devono condividere la CPU senza poterla utilizzare contemporaneamente. Ogni processo in genere ha un tempo di esecuzione stimato e una priorità associata, mentre la CPU ha a disposizione un certo numero di core, ciascuno dei quali può eseguire un solo processo per volta.

L'obbiettivo dello **scheduling dei processi** è allocare i processi ai core in modo ottimale, cercando di massimizzare il valore della priorità complessiva dei processi eseguiti in un dato periodo di tempo.

Lo scheduling può essere formulato come problema di programmazione lineare, risolvibile quindi da metodi come quello del simplesso, tramite solver come CPLEX. Tuttavia è anche possibile costruire un grafo G sulla base dei dati del problema, tale che la soluzione ottima di un problema di **Longest Path** su G coincida con la soluzione ottima del problema di scheduling originale.

In questa tesina verranno costruiti dei grafi a partire da alcune istanze del problema di scheduling dei processi, in modo da poter risolvere un Longest Path Problem e il relativo Shortest Path Problem tramite **CPLEX** e due noti algoritmi specifici per gli SPPs: il **Reaching Method** e il **Label Correcting modificato**, i cui risultati verranno confrontati con quelli del problema di scheduling originale.

Gli algoritmi sono stati scelti tenendo conto del fatto che i problemi di cammino minimo risultanti avranno costi degli archi negativi e saranno aciclici; inoltre si terrà conto di processori con un solo core a disposizione, dato che con più core la soluzione del problema non può essere rappresentata come un unico cammino lineare.

Cenni di teoria

Process scheduling originale

È noto che il problema di scheduling dei processi sia molto simile a quello di *knapsack* e possa essere formalizzato come segue, per poter essere risolto con i classici algoritmi di **Programmazione Lineare**:

- n: numero di processi da eseguire
- t_j: tempo di esecuzione richiesto dal processo *j*
- p_j: priorità del processo *j*
- T_{lim}: tempo a disposizione della CPU

Con le seguenti variabili decisionali binarie:

- $X_j \in \{0, 1\}$: vale 1 se il processo *j* viene eseguito (assegnato al core), altrimenti 0.

La **funzione obbiettivo** che punta a massimizzare il valore complessivo delle priorità dei processi eseguiti entro il tempo limite del processore:

$$max \sum_{j=1}^{n} xj * pj$$

I **vincoli** che impongono il rispetto del tempo totale del processore e il fatto che le variabili sono binarie:

$$\sum_{j=1}^{n} tj * xj \ \leq Tlim$$

$$xj \in \{0, 1\}, \qquad \forall j \in \{1, \dots, n\}$$

Come da premesse, attraverso questa formulazione è possibile costruire agevolmente un modello CPLEX per ottenere la soluzione ottima del problema.

Relazione con il Longest Path Problem

A partire dal problema precedentemente descritto è possibile costruire un grafo orientato e aciclico (**DAG**) con $n+2$ strati, di cui:

- Il primo strato contiene solo un nodo sorgente s
- L'ultimo strato contiene solo un nodo sink t
- Gli strati intermedi sono associati agli n processi

In particolare, negli strati associati agli n processi:

- Lo strato corrispondente al processo j rappresenta l'avanzamento nel considerare i primi j processi e contiene $T_{lim}+1$ nodi: $j^0, j^1, \ldots, j^{Tlim}$.
- Il nodo j^k rappresenta uno stato, in cui i processi $1, 2, \ldots, j$ considerati finora hanno consumato k unità di tempo in totale.
- Il nodo j^k ha al più 2 archi uscenti che rappresentano le decisioni di includere o non includere il processo $j+1$ nell'assegnamento al processore e hanno rispettivamente costi pari a p_{j+1} e 0.
- Ogni arco corrispondente alla decisione di includere un nuovo processo nell'assegnamento al core è presente solo se il tempo residuo è sufficiente.
- Tutti i nodi dello strato associato all'ultimo processo sono connessi al nodo t con archi di costo nullo.

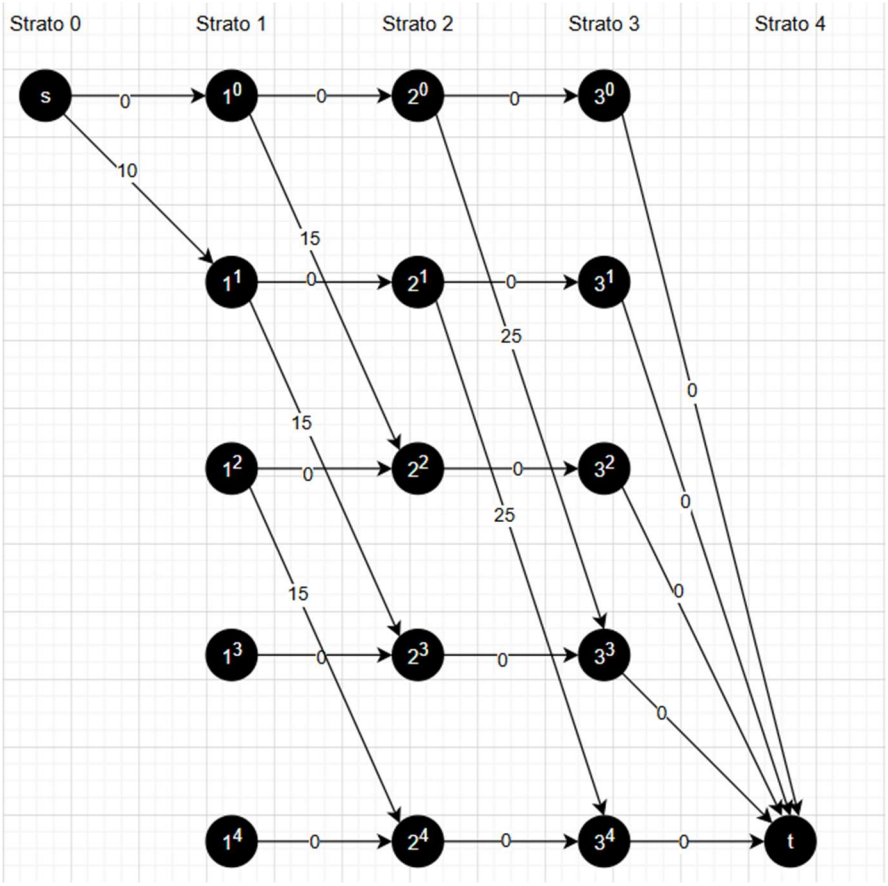
Un grafo così strutturato è rilevante dal punto di vista del nostro obbiettivo, dato che ogni cammino da **s** a **t** definisce una soluzione ammissibile del problema di scheduling originale e ha un costo pari alla priorità complessiva dei processi assegnati al processore. Allo stesso modo, ogni soluzione ammissibile del problema di scheduling originale corrisponde a un cammino diretto da **s** a **t**.

Di conseguenza trovare la soluzione ottima del problema di scheduling dei processi, in cui è massimizzata la priorità totale dei processi eseguiti entro il tempo limite del processore, equivale a cercare il **cammino massimo** da **s** a **t** nel **DAG** costruito come da specifiche, visto il modo in cui i costi sono associati agli archi sulla base delle priorità dei processi.

Per esempio, costruiamo il DAG corrispondente a un problema di scheduling su un core con $T_{lim}=4$ e 3 processi a disposizione con le seguenti caratteristiche:

j	1	2	3
p_j	10	15	25
t_j	1	2	3

La struttura risultante sarà la seguente:



Abbiamo n+2 strati: gli n strati associati ai processi che hanno Tlim+1 **nodi** che ricordano il tempo già impiegato, più i due strati della sorgente e del sink.

Abbiamo gli **archi** che rappresentano le scelte di eseguire o meno un processo, con un costo.

Ogni cammino diretto da **s** a **t** nel grafo in figura corrisponde a una soluzione ammissibile del problema di scheduling precedentemente descritto.

La soluzione ottima del problema originale consiste nella scelta dei processi **1** e **3** che massimizzano la priorità complessiva, con un valore della funzione obbiettivo pari a **35**.

Il cammino del grafo che corrisponde a questa soluzione è **s-1^1-2^1-3^4-t**, dato che rappresenta la scelta del processo 1 con l'arco **s-1^1**, l'esclusione del processo 2 con l'arco **1^1-2^1** e la scelta del processo 3 con l'arco **2^1-3^4**. Analizzando il grafo è semplice notare come **s-1^1-2^1-3^4-t** sia anche il cammino massimo tra **s** e **t**, come spiegato in precedenza.

Risolvere il problema di scheduling in questa maniera ricorda i concetti della **programmazione dinamica**, dato che abbiamo degli stati che riassumono le decisioni prese finora e ci muoviamo di stato in stato tramite delle decisioni che hanno un costo (che in questo caso deve essere massimizzato).

Formalizzazione Longest Path Problem

Anche la ricerca di un cammino massimo da *s* a *t* su un grafo può essere formulata come un problema di programmazione lineare con vincoli identici a un problema di **Shortest Path** da sorgente a sink, ma con obbiettivo "opposto", dato che nel LPP stiamo cercando di massimizzare i costi, mentre nello SPP puntiamo a minimizzarli.

La **funzione obbiettivo** punta a massimizzare la somma dei costi lungo il cammino da *s* a *t*:

$$\boldsymbol{max} \sum_{(\boldsymbol{i},\boldsymbol{j})\in \boldsymbol{A}} \boldsymbol{xij} * \boldsymbol{cij}$$

Dove ricordiamo che i costi degli archi del nostro DAG corrispondono ai costi delle decisioni di includere o meno i processi nell'assegnamento al processore, e hanno quindi costi nulli o pari alle **priorità** associate ai processi.

I vincoli di **conservazione del flusso** variano in base ai nodi:

$$\sum_{(\boldsymbol{i},\boldsymbol{j})\in \boldsymbol{\delta}+(\boldsymbol{i})} \boldsymbol{xij} - \sum_{(\boldsymbol{i},\boldsymbol{j})\in \boldsymbol{\delta}-(\boldsymbol{i})} \boldsymbol{xji} = \mathbf{1} \qquad \boldsymbol{se\ i = s}$$

$$\sum_{(\boldsymbol{i},\boldsymbol{j})\in \boldsymbol{\delta}+(\boldsymbol{i})} \boldsymbol{xij} - \sum_{(\boldsymbol{i},\boldsymbol{j})\in \boldsymbol{\delta}-(\boldsymbol{i})} \boldsymbol{xji} = -\mathbf{1} \qquad \boldsymbol{se\ i = t}$$

$$\sum_{(\boldsymbol{i},\boldsymbol{j})\in \boldsymbol{\delta}+(\boldsymbol{i})} \boldsymbol{xij} - \sum_{(\boldsymbol{i},\boldsymbol{j})\in \boldsymbol{\delta}-(\boldsymbol{i})} \boldsymbol{xji} = \mathbf{0} \qquad \forall \boldsymbol{i} \in \boldsymbol{N} - \{\boldsymbol{s},\boldsymbol{t}\}$$

Dove ***δ+*** e ***δ-*** sono rispettivamente gli insiemi degli archi uscenti ed entranti del nodo i.

Dalla ricerca di un cammino da *s* a *t*, infatti ci aspettiamo che:

- Un solo arco selezionato esca da s
- Un solo arco selezionato entri in t
- Gli altri nodi abbiano un arco selezionato in entrata e uno in uscita, oppure nessun arco selezionato incidente.

Infine abbiamo il vincolo sulle **variabili decisionali**, che sono binarie e assumono valore 1 se l'arco corrispondente è incluso nel cammino selezionato.

$$\boldsymbol{xij} \in \{\mathbf{0}, \mathbf{1}\}, \qquad \forall(\boldsymbol{i}, \boldsymbol{j}) \in \boldsymbol{A}$$

Un problema di Longest Path formulato in questo modo è risolvibile anche con il metodo del simplesso, quindi con molti solver generici di programmazione lineare, tra cui CPLEX, come verrà mostrato in seguito.

Shortest Path Problem e algoritmi label correcting

Una volta ottenuto il grafo dai dati del problema di scheduling, oltre a risolvere un problema di Longest Path, è possibile invertire il segno dei costi degli archi per poter risolvere un problema di **Shortest Path**, sfruttando così gli algoritmi specifici.

Questa operazione è possibile nel nostro caso, perché trattandosi di un grafo aciclico, si ha la certezza che, anche con costi degli archi negativi, non avremmo mai un **ciclo negativo** che determinerebbe una soluzione illimitata a causa di una funzione obbiettivo sempre migliorabile.

Tuttavia bisogna comunque fare attenzione nella scelta degli algoritmi, dato che alcuni di questi, come il famoso algoritmo di **Dijkstra**, non possono essere utilizzati su grafi con costi degli archi negativi, come il DAG costruito sulla base del nostro problema di scheduling.

Per risolvere problemi di cammino minimo esistono algoritmi label setting e label correcting, classificati in base a come gestiscono le **"distance labels"** dei nodi, che tengono traccia del limite superiore delle distanze dei nodi dalla sorgente:

- Gli algoritmi **label setting**, a ogni step scelgono un nodo *temporaneo* (la cui distance label non è ancora stata provata ottima), provano che questo nodo sia effettivamente *permanente* (la cui distance label è di comprovata ottimalità) e aggiornano le distance labels di conseguenza.
- Gli algoritmi **label correcting** invece considerano tutte le labels temporanee fino all'ultima iterazione.

Il primo algoritmo utilizzato per risolvere lo Shortest Path Problem di interesse è il **label correcting modificato**.

In generale gli algoritmi label correcting ad ogni step individuano un arco che viola le **condizioni di ottimalità**:

$$\boldsymbol{d(j)} \leq \boldsymbol{d(i)} + \boldsymbol{cij}, \qquad \forall \boldsymbol{j} \in \boldsymbol{N}$$

E correggono **d(j)**, che è la **distance label** del nodo **j**, che indica l'upper bound corrente della distanza di **j** dal nodo sorgente.

Infatti nel caso in cui:

$$\boldsymbol{d(j) > d(i) + cij}, \qquad \forall \boldsymbol{j} \in \boldsymbol{N}$$

La corrente distance label di **j** potrebbe essere migliorata scegliendo **i** come predecessore di **j**.

Il Modified Label Correcting funziona in questo modo:

- Inizializza la distance label di ogni nodo a infinito
- Pone la distance label e il predecessore del nodo sorgente **s** a 0
- Crea una lista **LIST** in cui inserisce inizialmente solo **s**
- Finché **LIST** non è vuota:
    - Rimuove un elemento **i** da **LIST**
    - Se qualche arco **(i,j)** viola le condizioni di ottimalità, quindi se **$d(j)>d(i)+c_{ij}$**:
        - Aggiorna la distance label di **j**, quindi **$d(j)=d(i)+c_{ij}$**
        - Aggiorna il predecessore di **j**, che diventa **i**
        - Se **j** non è già in **LIST**, lo aggiunge

Gli algoritmi label correcting, come quello appena descritto, possono essere utilizzati nel caso di grafi con costi degli archi arbitrari e cicli (a patto che non siano negativi), tuttavia hanno una complessità computazionale abbastanza elevata: per il label correcting modificato, nel caso peggiore si può arrivare a ***O(nmC)***, dove ***n*** è il numero dei nodi, ***m*** il numero degli archi e ***C*** il massimo valore dei costi a cui può arrivare un cammino ottimo.

Algoritmi label setting

Gli algoritmi label setting hanno costi computazionali decisamente minori, ma non possono essere utilizzati in tutti i casi.

L'altro algoritmo con cui verrà risolto il nostro problema di Shortest Path è il **reaching method**: un algoritmo label setting che può essere applicato su grafi aciclici con costi degli archi arbitrari.

Il reaching method ha bisogno di conoscere un **ordine topologico** dei nodi del grafo (che esiste sicuramente nei grafi aciclici come il nostro), per esempio un array ***order***, tale che:

$$\boldsymbol{order(i) < order(j)}, \qquad \forall (\boldsymbol{i}, \boldsymbol{j}) \in \boldsymbol{A}$$

Dopo aver ottenuto un ordine topologico del grafo, il reaching method a ogni step seleziona un nodo **i** seguendo l'ordine topologico e aggiorna le distance labels dei nodi adiacenti **j** se è conveniente raggiungerli da **i**.

Nello specifico:

- Ordina topologicamente i nodi
- Inizializza la distance label di ogni nodo a infinito
- Pone la distance label e il predecessore del nodo sorgente **s** a 0
- Per ogni nodo **i** in ordine topologico:
    - Per ogni nodo **j** adiacente:
        - Se $\mathbf{d(j)>d(i)+c_{ij}}$
        - Aggiorna la distance label di **j**, quindi $\mathbf{d(j)=d(i)+c_{ij}}$
        - Aggiorna il predecessore di **j**, che diventa **i**

Per ottenere un ordine topologico dei nodi si utilizza un algoritmo che sfrutta la proprietà per cui i grafi aciclici hanno sempre almeno un nodo con indegree pari a zero.

Di conseguenza, a ogni iterazione:

- Finché il grafo non è vuoto:
    - Seleziona un nodo **i** con indegree=0
    - Lo inserisce nell'ordine
    - Rimuove **i** e gli archi incidenti a **i** dal grafo

La complessità computazionale del reaching method sarebbe lineare al numero degli archi, tuttavia tenendo conto della procedura di ordinamento topologico la complessità totale corrisponde a $\boldsymbol{O(n+m)}$.

In caso di grafi aciclici con costi arbitrari si potrebbe utilizzare anche il **pulling method**, che al posto di selezionare **i** in ordine topologico, seleziona **j**; tuttavia utilizzando *le liste di adiacenza* per contenere i dati del problema è più indicato il reaching method appena descritto, dato che non c'è bisogno di calcolare liste di adiacenza inverse o di conoscere i predecessori dei nodi.

Implementazione

Generazione dati

Per iniziare è stata riutilizzata una funzione che genera un set di dati su cui risolvere il problema originale di scheduling dei processi:

- Il tempo limite di un core
- Un insieme di processi con i relativi tempi di esecuzione e valori di priorità

Sulla base di:

- Numero di core posto a 1
- Numero di processi da eseguire
- Tempo limite dei core
- Tempo di esecuzione massimo dei processi generati casualmente

La generazione dei dati è pseudo-randomica e il seme preimpostato ci permette di ripetere gli esperimenti con lo stesso set di dati in modo da poter confrontare i diversi algoritmi.

```
def generaDati(numeroCore, numeroProcessi, tempoCore, tempoMaxProcessi):

    np.random.seed(88)
    Tlim= [tempoCore] * numeroCore
    t = np.random.randint( low: 1, tempoMaxProcessi, numeroProcessi)  # Tempi di esecuzione dei processi
    p = np.random.randint( low: 1,  high: 10, numeroProcessi)  # Priorità dei processi
    nomi = [f'processo_{i}' for i in range(numeroProcessi)]

    dati = {
        "Tlim": Tlim,
        "processi": [
            {"nome": nomi[i], "t": t[i], "p": p[i]} for i in range(numeroProcessi)
        ]
    }

    return dati
```

È stata inclusa nel codice anche una funzione per poter definire piccoli insiemi di dati con valori arbitrari, a scopo di test:

```
def datiProva():
    dati = {
        "Tlim": [70],  # Tempo limite per il core
        "processi": [
            {"nome": "processo_1", "t": 15, "p": 8},  # Processo 1: tempo 15, priorità 8
            {"nome": "processo_2", "t": 20, "p": 5},  # ecc
            {"nome": "processo_3", "t": 10, "p": 2},
            {"nome": "processo_4", "t": 25, "p": 7}
        ]
    }

    return dati
```

Generazione grafo

È stata scritta una funzione che genera un grafo a partire dai dati del problema di scheduling (tempo del core e insieme di processi), seguendo sostanzialmente le specifiche dei paragrafi di teoria:

- Vengono creati inizialmente tanti strati quanti gli **n** processi
- Ogni strato contiene $\mathbf{T_{lim}+1}$ nodi
- Vengono creati gli archi che rappresentano le decisioni di inclusione dei processi, con i costi nulli o pari alla priorità del processo in questione
- Vengono creati i nodi **source** e **sink**, che corrispondono agli ultimi 2 strati

Agli archi sono stati assegnati anche dei nomi che iniziano con **yes** o **no**, per poter risalire alle decisioni prese dagli algoritmi al termine dell'esecuzione e i loro costi sono chiamati *"priority"*, per consistenza con il problema originale.

I nodi sono rappresentati come delle coppie **(i,k)**, per tenere traccia sia dello strato (processo) **i**, che delle unità di tempo **k** già consumate nello stato corrispondente.

Ogni arco contiene le informazioni di nodi incidenti, priorità e decisione corrispondente.

```
#Crea il grafo a partire dai dati del problema di scheduling
archi = []
for i in range(n ):        #tanti strati quanti processi per adesso
    for k in range(T + 1):        #nodi per strato, rappresentanti k istanti consumati
        if i < n:                    #creazione nodi
            dur = processes[i]["t"]
            pr = processes[i]["p"]
            nm = processes[i]["nome"]

            #arco scelta di non eseguire il processo
            archi.append({"from": (i, k), "to": (i + 1, k), "priority": 0, "dec": f"no_{nm}"})

            #arco scelta di eseguire il processo, se il tempo residuo lo permette
            if k + dur <= T:
                archi.append({
                    "from": (i, k), "to": (i+1, k + dur),
                    "priority": pr, "dec": f"yes_{nm}"
                })

# aggiunge source e sink e li collega ai nodi opportuni
source, sink = "s", "t"
archi.append({"from": source, "to": (0, 0), "priority": 0, "dec": "start"})
for k in range(T + 1):
    archi.append({"from": (n, k), "to": sink, "priority": 0, "dec": f"end_{k}"})
```

Per gli algoritmi di Shortest Path, il costo degli archi sarà uguale al negativo della priorità.

Creazione modello CPLEX

Come primo metodo per risolvere il problema di longest path sui grafi creati al passaggio precedente è stato scelto un modello CPLEX, puramente per confrontare le soluzioni ottime della formulazione di PL sul grafo con quelle del problema di scheduling originale.

Utilizzare questo metodo per risolvere un problema di questo tipo infatti è abbastanza svantaggioso, anche per i tempi di esecuzione del modello, ma soprattutto per i tempi di costruzione di quest'ultimo.

Per la generazione del modello è stata seguita la formulazione del LPP del paragrafo di teoria.

Per prima cosa è stata inserita la **funzione obbiettivo**, che punta a massimizzare i costi (le priorità) associati agli archi scelti nel cammino:

```
#Obiettivo: massimizza la somma delle priorità
model.maximize(model.sum(e["priority"] * x[(e["from"], e["to"], e["dec"])] for e in archi))
```

In seguito sono stati aggiunti i vincoli di **conservazione del flusso**, specifici per la sorgente, il sink e il resto dei nodi intermedi. Ricordiamo che il nodo sorgente deve avere un solo arco in uscita selezionato, il nodo sink un solo arco in entrata e tutti gli altri nodi devono avere un numero di archi in entrata pari al numero di archi in uscita:

```
#Vincoli di flow conservation
# uscita da source
model.add_constraint(sum(x[k] for k in x if k[0] == source) == 1)

# ingresso a sink
model.add_constraint(sum(x[k] for k in x if k[1] == sink) == 1)

# ogni altro nodo
nodes = set([e["from"] for e in archi] + [e["to"] for e in archi])
nodes.discard(source)
nodes.discard(sink)          #senza contare source e sink
for node in nodes:
    model.add_constraint(
        sum(x[k] for k in x if k[1] == node) == sum(x[k] for k in x if k[0] == node),
        ctname: f"flow_{node}"
    )
```

È possibile esaminare i flussi in entrata e in uscita dai nodi, dato che i vincoli sono stati nominati in base a questi ultimi.

Infine è stato imposto il vincolo sulle **variabili decisionali** associate agli archi, che sappiamo essere **binarie** e che sono state nominate sulla base dei nodi incidenti e sulla decisione che rappresentano:

```
#Variabili binarie per arco
x = {}
for e in archi:
    key = (e["from"], e["to"], e["dec"])
    x[key] = model.binary_var(name=f"x_{e['dec']}_{e['from']}_{e['to']}")
```

Quindi, dato un problema di scheduling dei processi e il grafo ottenuto dai suoi dati, ci si aspetta che le soluzioni ottime del problema originale risolto con CPLEX e del problema di Longest Path sul grafo risolto con il modello appena descritto coincidano.

Modified label correcting

Un metodo più efficiente per risolvere questo problema consiste sicuramente nell'invertire il segno dei costi degli archi del grafo, per poter risolvere un problema di cammino minimo, sfruttando gli algoritmi specifici più efficienti. La motivazione della scelta degli algoritmi è stata spiegata in precedenza ed è legata alla negatività dei costi degli archi nel grafo aciclico ottenuto.

Il primo algoritmo proposto per risolvere il problema di Shortest Path ottenuto invertendo il segno dei costi degli archi del DAG generato dal problema di scheduling è il **Modified Label Correcting**, implementato come dalle specifiche del rispettivo paragrafo di teoria.

In seguito alla costruzione della struttura del grafo, vengono inizializzate le distance labels, i predecessori, la distanza del nodo sorgente e la lista LIST:

```
#Inizializza
d = {nodo: float('inf') for nodo in N}
pred = {nodo: None for nodo in N}
d[source] = 0

LIST = deque([source])
```

In seguito inizia l'algoritmo vero è proprio, che aggiorna la lista, controlla l'ottimalità degli archi, corregge le distance labels e aggiorna i predecessori:

```
while LIST:
    i = LIST.popleft()
    for j, cij in A[i]:
        if d[j] > d[i] + cij:
            d[j] = d[i] + cij
            pred[j] = i
            if j not in LIST:
                LIST.append(j)

return d[sink], pred
```

Vengono infine restituiti:

- La distanza del nodo sink, che corrisponde al valore della funzione obbiettivo del problema di scheduling originale.
- La lista dei predecessori che identificano il cammino individuato.

Reaching method

Il label correcting modificato è sicuramente più efficiente del modello CPLEX per il Longest Path, tuttavia, come anticipato, il modo migliore per risolvere problemi di cammino minimo è utilizzare algoritmi label setting, quando possibile. Il secondo algoritmo proposto è il **reaching method**, implementato come da specifiche.

In seguito alla costruzione del grafo, viene eseguita la procedura di ordinamento topologico, che calcola gli indegree iniziali dei nodi, utilizza una lista per tenere traccia dei nodi con indegree=0 e ad ogni step rimuove un nodo dalla lista, lo aggiunge all'ordine e aggiorna gli indegree:

```
# Calcolo indegree
inDegree = defaultdict(int)
for i in A:
    for j, _ in A[i]:
        inDegree[j] += 1
    if i not in inDegree:
        inDegree[i] = 0  # anche nodi senza archi entranti

#ordinamento topologico
LIST = [node for node in N if inDegree[node] == 0]
order = []

while LIST:
    i = LIST.pop(0)
    order.append(i)
    for j, _ in A[i]:
        inDegree[j] -= 1
        if inDegree[j] == 0:
            LIST.append(j)
```

In seguito vengono inizializzate distanze e predecessori:

```
#inizializza
d = {node: float('inf') for node in N}
pred = {node: None for node in N}
d[source] = 0
```

E infine l’algoritmo effettivo aggiorna le distance labels dei nodi, procedendo in ordine topologico:

```
for i in order:
    for j, cij in A[i]:
        if d[j] > d[i] + cij:
            d[j] = d[i] + cij
            pred[j] = i

return d[sink], pred
```

Vengono restituiti anche il questo caso la distanza ottima del nodo sink e la lista dei predecessori.

Test

Prima di osservare il comportamento degli algoritmi e dei modelli su diversi problemi di scheduling, si è ritenuto opportuno verificare la correttezza degli algoritmi su un dataset di dimensioni ridotte:

```
"Tlim": [70],  # Tempo limite per ciascun core
"processi": [
    {"nome": "processo_0", "t": 15, "p": 8},  # Processo 1: tempo 15, priorità 8
    {"nome": "processo_1", "t": 20, "p": 6},  # Processo 2: tempo 20, priorità 5
    {"nome": "processo_2", "t": 10, "p": 2},  # E cosi via
    {"nome": "processo_3", "t": 25, "p": 7},
    {"nome": "processo_4", "t": 30, "p": 6},
    {"nome": "processo_5", "t": 3, "p": 8},
    {"nome": "processo_6", "t": 18, "p": 5},
    {"nome": "processo_7", "t": 40, "p": 10},
    {"nome": "processo_8", "t": 25, "p": 3},
    {"nome": "processo_9", "t": 30, "p": 9}
]
```

Per prima cosa è stato risolto il normale **problema di scheduling** con un modello CPLEX che punta a massimizzare la priorità complessiva dei processi eseguiti entro il tempo limite (tesina di Decision Science), che ha prodotto i seguenti risultati:

```
objective: 31
status: OPTIMAL_SOLUTION(2)
  x_0_0=1
  x_0_1=1
  x_0_5=1
  x_0_9=1
```

Indicando che la priorità complessiva totale ottima è pari a **31** e si ottiene selezionando i processi **0, 1, 5, 9**.

In seguito è stato generato il grafo a partire da questi dati, secondo il metodo spiegato in precedenza ed è stato risolto il **problema di Longest Path** tramite il modello CPLEX implementato nel capitolo scorso, producendo i risultati:

```
objective: 31
status: OPTIMAL_SOLUTION(2)
  "x_yes_processo_0_(0, 0)_(1, 15)"=1
  "x_yes_processo_1_(1, 15)_(2, 35)"=1
  "x_no_processo_2_(2, 35)_(3, 35)"=1
  "x_no_processo_3_(3, 35)_(4, 35)"=1
  "x_no_processo_4_(4, 35)_(5, 35)"=1
  "x_yes_processo_5_(5, 35)_(6, 38)"=1
  "x_no_processo_6_(6, 38)_(7, 38)"=1
  "x_no_processo_7_(7, 38)_(8, 38)"=1
  "x_no_processo_8_(8, 38)_(9, 38)"=1
  "x_yes_processo_9_(9, 38)_(10, 68)"=1
  "x_start_s_(0, 0)"=1
  "x_end_68_(10, 68)_t"=1
```

Il valore della funzione obbiettivo è sempre pari a **31** e le variabili relative agli archi selezionati ci permettono di interpretare le scelte corrispondenti, che corrispondono a quelle del modello precedente di eseguire i processi **0, 1, 5, 9**.

Questo risultato ci conferma che la conversione del problema di scheduling in LPP è corretta e il modello funziona come dovrebbe.

Tuttavia come accennato in precedenza i **tempi di costruzione** del modello sono maggiori per il Longest Path Problem:

Problema	Scheduling originale	LPP
Tempo costruzione	0.002371 s	0.159002 s
Tempo esecuzione	0.016 s	0.016 s

Questa differenza cresce con le dimensioni del problema ed è dovuta chiaramente al fatto che, formulando il problema come grafo a livelli, il modello risolutivo ha bisogno di molte più variabili e molti più vincoli.

Per questo motivo è più vantaggioso convertire il Longest Path Problem in **Shortest Path Problem** e risolverlo con gli algoritmi specifici per questo tipo di problemi.

Eseguendo il **Label Correcting Modificato** per lo SPP basato sui dati del problema di scheduling si ottiene il medesimo risultato dei modelli precedenti, che conferma la correttezza delle trasformazioni del problema e dell'algoritmo:

```
Priorità totale ottima: 31
Processi selezionati:
 - processo_0 (priorità 8, durata implicita)
 - processo_1 (priorità 6, durata implicita)
 - processo_5 (priorità 8, durata implicita)
 - processo_9 (priorità 9, durata implicita)
```

Anche eseguendo il **Reaching Method** si ottengono gli stessi risultati:

```
Priorità totale ottima: 31
Processi selezionati:
 - processo_0 (priorità 8, durata implicita)
 - processo_1 (priorità 6, durata implicita)
 - processo_5 (priorità 8, durata implicita)
 - processo_9 (priorità 9, durata implicita)
```

Osservando i tempi di esecuzione non si notano particolari vantaggi dell'algoritmo label setting per adesso:

Algoritmo	LC Modificato	Reaching Method
Tempo esecuzione	0.000994 s	0.001771 s

Probabilmente la differenza si noterà maggiormente con l'aumentare delle dimensioni dei problemi.

Risultati e conclusioni

Per confermare i risultati dell'esempio su dataset ristretto e per confrontare il comportamento degli algoritmi, sono stati condotti dei test su insiemi di dati generati casualmente a partire da questi parametri:

NOME	SEED	PROCESSI	Tlim	Tmax PROCESSO
s1	11	50	100	6
s2	19	100	220	8
s3	23	100	270	9
s4	38	200	500	12
s5	48	200	400	10
s6	59	200	500	7
s7	65	250	500	11
s8	76	250	550	12
s9	88	250	600	12
s10	93	250	700	11

Naturalmente in ogni caso sia i due modelli CPLEX che i due algoritmi per gli SPP hanno individuato la soluzione **ottima** del problema, quindi verranno confrontati solo i tempi di esecuzione e di costruzione dei modelli (per i modelli CPLEX).

Risolvere un problema di Longest Path con un modello CPLEX su questo tipo di grafi a strati è molto svantaggioso come anticipato, infatti semplicemente aumentando il numero di processi a 100, i **tempi di costruzione** in **secondi** del modello LPP lievitano notevolmente rispetto a quelli per costruire il semplice modello in stile knapsack:

DATI	CPLEX SCHEDULING	CPLEX LPP
s1	0.010377	5.659405
s2	0.006395	133.71798

Il motivo di questa dinamica è il modo in cui variano i numeri di **variabili** e **vincoli** nel problema sul grafo, infatt:

- Nel modello stile knapsack il numero di vincoli e variabili cresce linearmente rispetto al numero di processi ***n***.
- Nel grafo su cui risolviamo il problema di Longest Path invece abbiamo **n** strati con $\mathbf{T_{lim}+1}$ nodi ciascuno, a cui aggiungere sorgente e sink, quindi il numero di variabili associate agli archi e di conseguenza il numero dei vincoli aumentano linearmente rispetto a $\boldsymbol{nT}$, rendendo il modello molto più complesso.

Per ragioni legate al tempo di costruzione del modello per LPP, si è deciso di proseguire i test solo con gli altri algoritmi, conducendo ai seguenti **tempi di esecuzione** in **secondi**:

DATI	CPLEX SCHEDULING	CPLEX LPP	LABEL CORRECTING	REACHING METHOD
s1	0.030999	0.108999	0.019673	0.020544
s2	0.031	0.75	0.168453	0.130551
s3	0.032		0.247995	0.185238
s4	0.016		1.607334	1.062372
s5	0.016		1.073571	1.025973
s6	0.016		1.25164	0.8631
s7	0.016		2.223915	1.089296
s8	0.016		2.550129	1.287466
s9	0.015		3.201146	1.317433
s10	0.016		3.538658	1.60241

Come si era ipotizzato, il reaching method è più efficiente del label correcting modificato man mano che la dimensione del problema cresce, tuttavia entrambi gli algoritmi non sono paragonabili alla risoluzione del problema di scheduling originale trattato come un knapsack problem.

Conclusioni

Osservando i risultati degli esperimenti condotti è possibile trarre le seguenti conclusioni:

- I problemi di scheduling di processi sono trattabili come problemi di **Longest Path**, costruendo un grafo aciclico per tenere conto delle possibili soluzioni ammissibili, viste come un insieme di stati e decisioni.
- Tuttavia risolvere un LPP direttamente con un modello CPLEX non conviene. È meglio invertire il segno degli archi del grafo e risolvere un problema di **Shortest Path** con gli algoritmi specifici esistenti.
- Il **reaching method** si conferma più efficiente del **label correcting modificato**, tuttavia gli algoritmi label setting non possono essere applicati in tutti gli scenari.
- Risolvere il problema di scheduling dei processi direttamente con un modello CPLEX in stile **knapsack**, senza creare alcun grafo, rimane comunque la scelta migliore.

In particolare si evidenzia l'applicabilità degli algoritmi di Shortest Path anche su problemi che apparentemente non hanno nulla a che fare con grafi o reti.