UNIVERSITÀ
DEGLI STUDI
DI PADOVA

DIPARTIMENTO
DI INGEGNERIA
DELL'INFORMAZIONE

CORSO DI LAUREA IN INGEGNERIA INFORMATICA

MODELLI MIP PER PROBLEMI BASATI SUL NUMERO DI ALCUIN DI UN GRAFO

Relatore: Prof. Domenico Salvagnin

Laureando: Michele Sprocatti

ANNO ACCADEMICO: 2022-2023

Data di laurea: 29 settembre 2023

Ringraziamenti

Ringrazio il mio relatore per la disponibilità e il supporto fornito durante la stesura di questo elaborato.

Ringrazio i miei genitori, Giuseppina e Giampaolo e i miei fratelli Matteo e Andrea, che mi hanno sempre sostenuto, appoggiando ogni mia decisione, fin dalla scelta del mio percorso di studi.

Ringrazio la mia fidanzata Giulia perchè mi ha sempre sostenuto e sopportato in tutti questi anni. Grazie per tutti i tuoi consigli e per il tuo aiuto. Grazie perchè ci sei sempre stata.

Ringrazio tutti i miei parenti: nonni, zii e cugini per il loro aiuto e i loro consigli durante il percorso di studi. Ringrazio anche mia zia Ornella che sarebbe orgogliosa di vedermi qui oggi.

Ringrazio i miei amici per essere stati sempre presenti durante questa fase del mio percorso di studi.

Abstract

Il presente lavoro di tesi si prefigge lo studio di una generalizzazione dell'Alcuin river crossing problem, noto anche come il problema dell'attraversamento del fiume di Alcuin, con l'obiettivo di analizzare i tempi necessari per determinare la soluzione del problema e di valutare il legame tra il numero di Alcuin di un grafo e la sua densità.
Questo elaborato è stato articolato in 4 capitoli: inizialmente si presentano alcune nozioni teoriche fondamentali, necessarie per la comprensione dello studio, successivamente si presentano le modalità usate per affrontare le prove sperimentali e si analizzano i risultati ottenuti. Infine si elaborano le conclusioni del lavoro svolto.

Per affrontare questo studio sono stati scritti diversi script Python[1] che svolgono le varie funzioni necessarie partendo dalla generazione casuale di grafi fatta usando il pacchetto python Netwokx[2], fino all'elaborazione dei risultati ottenuti. Tutto il codice è consultabile liberamente online tramite la repository usata per il controllo versione del progetto[14]. Per la risoluzione dei modelli matematici è stato usato il software CPLEX 22.1.1[9] e la sua interfaccia su Python[1] tramite il pacchetto Pyomo[3]. Per la presentazione dei risultati in maniera grafica e per l'elaborazione del file .csv sono stati utilizzati i seguenti pacchetti Python[1]: Matplotlib[4], Seaborn[5], Numpy[6], Pandas[7].
I tempi di calcolo presenti nei risultati di questo elaborato fanno riferimento al cluster del Dipartmento di Ingegneria dell'Informazione dell'Università degli Studi di Padova. Per poter usare il cluster è stato necessario l'utilizzo dello scheduler generico SLURM[8].

Molte delle definizioni presenti in questo elaborato sono ispirate alle dispense del corso di Modelli e Software per l'Ottimizzazione Discreta, tenuto dal professore Domenico Salvagnin[11][12][13] presso l'Università degli Studi di Padova.

Indice

Capitolo 1

Nozioni preliminari

La ricerca operativa è una parte della matematica applicata che si occupa di risolvere problemi decisionali complessi tramite l'utilizzo di modelli matematici. Essa riveste un ruolo importante nelle attività decisionali dato che permette di operare le scelte migliori per raggiungere un determinato obiettivo, rispettando vincoli che sono imposti dall'esterno e non dipendono dal controllo di chi deve compiere le decisioni. Un campo della ricerca operativa è l'ottimizzazione, questa si occupa di problemi in cui bisogna massimizzare o minimizzare una funzione sottoposta a dei vincoli. Il processo di risoluzione di un dato problema segue questi step:

1. Analisi del problema nel mondo reale
2. Scrittura del modello matematico
3. Risoluzione del problema ottenendo la soluzione ottima, se c'è, ed interpretarla per riportarla al mondo reale

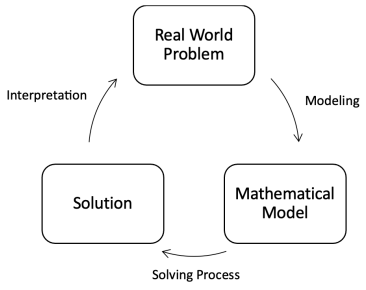


Figura 1.1: Optimization scheme

1.1 Problemi di ottimizzazzione

Un problema di ottimizzazione P si può formulare come:

$$\begin{cases} min(or\ max)\ f(x) \\ S \\ x \in D \end{cases} \tag{1.1}$$

dove

1. f(x) è una funzione a valore reali nelle variabili x
2. D è il dominio di x
3. S è un insieme finito di vincoli

In generale, x è una tupla $(x_1, \ldots, x_n)$ e D è il prodotto cartesiano $D_1 \times \cdots \times D_n$, e vale $x_j \in D_j$.
Un vincolo $c \in S$ è una funzione associata ad un sottoinsieme di variabili x_c. Esso può essere soddisfatto o violato.
Ogni $x \in D$ si dice soluzione di P. Una soluzione che soddisfi tutti i vincoli in S si dice ammissibile. L'insieme delle soluzioni ammissibili di un problema di ottimizzazione P si indica con F(P). Una soluzione ammissibile x^* si dice ottima se

$$\begin{array}{c} f(x^*) \leq f(x)\ \forall x \in F(P)\ (nel\ caso\ di\ problema\ di\ minimo) \\ f(x^*) \geq f(x)\ \forall x \in F(P)\ (nel\ caso\ di\ problema\ di\ massimo) \end{array} \tag{1.2}$$

Un problema di ottimizzazione si dice impossibile (infeasible) se non ammette alcuna soluzione ammissibile, cioè F(P) = $\emptyset$. Invece si dice illimitato (unbounded) se non esiste alcun limite inferiore a f(x) per $x \in F(P)$.
In particolare un problema di ottimizzazione nella forma (1.1) risulta intrattabile, quindi si considerano casi particolari più facili da risolvere, nel caso di studio di questo elaborato si prende in considerazione il caso di Programmazione lineare intera (MIP), ma per comprendere bene l'algoritmo risolutivo viene spiegato anche il caso di Programmazione lineare.

1.2 Programmazione lineare

Un problema di programmazione lineare consiste nella minimizzazione di una funzione lineare soggetta ad un numero finito di vincoli lineari. In generale si ha

la forma:

$$\begin{cases} min\ cx \\ a_i x \sim b_i \ \ i = 1, \ldots, m \\ l_j \leq x_j \leq u_j \ \ j = 1, \ldots, n \end{cases} \tag{1.3}$$

dove

1. $\sim \in \{\leq, \geq, =\}$
2. $l_j \in R \cup \{-\infty\}$
3. $u_j \in R \cup \{+\infty\}$

Quindi i domini delle singole x_j sono intervalli di R.

1.2.1 Interpetrazione geometrica

Ogni vincolo si può interpretare come la definizione di un iperpiano o semispazio affine, la loro intersezione genera un poliedro, che racchiude l'insieme delle soluzioni ammissibili del problema. Inoltre è dimostrato che: se il problema di programmazione lineare $min\{\ cx\ :\ x \in P\}$ ammette ottimo finito, allora esiste un vertice di P ottimo.

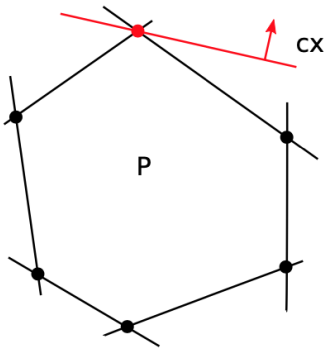


Figura 1.2: Interpretazione geometrica

1.2.2 Algoritmi risolutivi

Esistono 2 algoritmi risolutivi per i problemi di programmazione lineare:

1. Algoritmo del Simplesso: partendo da un vertice qualsiasi, ad ogni iterazione si sposta da un vertice ad un altro, fino a trovare il vertice ottimo.

2. Algoritmo Punto Interno/Barriera: partendo da un punto interno al poliedro, sfrutta il metodo di Newton per tracciare una sequenza di punti che porta al vertice ottimo.

1.3 Programmazione Lineare Intera

Un problema di programmazione lineare intera (MIP, Mixed Integer Programming) consiste nella minimizzazione di una funzione lineare soggetta ad un numero finito di vincoli lineari, con in più il vincolo che alcune variabili devono assumere valori interi. In generale si ha:

$$\begin{cases} min\ cx \\ a_i x \sim b_i \ \ i = 1, ..., m \\ l_j \leq x_j \leq u_j \ \ j = 1, ..., n = N \\ x_j \in Z \ \ \forall j \in J \subseteq N = \{1, ..., n\} \end{cases} . \tag{1.4}$$

Dove

1. $\sim \in \{\leq, \geq, =\}$
2. $l_j \in R \cup \{-\infty\}$
3. $u_j \in R \cup \{+\infty\}$

Se J = N, si parla di programmazione lineare intera pura, altrimenti di programmazione lineare intera mista. Se invece J = $\emptyset$ si ha un semplice problema di programmazione lineare.
I problemi MIP possono essere risolti tramite tre algoritmi:

1. Branch & Bound (B&B)
2. Cutting Planes
3. Branch & Cut (B&C)

1.3.1 Algoritmo Branch & Bound (B&B)

L'algoritmo di Branch & Bound usa la strategia *divide and conquer* per dividere lo spazio di ricerca del problema in sottoproblemi, per poi ottimizzarli separatamente. Inoltre mediante il *pruning (by optimality o by infeasibility)* permette di eliminare intere parti dello spazio di ricerca, dove si è certi di non trovare soluzioni migliori dell'attuale.
Sia F l'insieme delle soluzioni ammissibili di un problema di minimizzazione (simmetrico nel caso della massimizzazione, a meno di un cambio di segno della funzione obiettivo), sia f : F $\rightarrow$ R la funzione obiettivo da minimizzare e $\bar{x} \in$ F una soluzione ammissibile nota, che prende il nome di incumbent, determinata ad esempio euristicamente. Il costo z = f($\bar{x}$) di tale soluzione ammissibile, è un upper bound sul costo della soluzione ottima. L'algoritmo B&B segue questi step:

1. Nella fase di bounding, si rilassa il problema, ammettendo soluzioni che appartengono ad un sovrainsieme di F. La soluzione del rilassamento fornisce pertanto un lower bound sul valore della soluzione ottima. Se la soluzione del rilassamento appartiene a F o ha costo uguale a z, allora l'algoritmo termina: ha trovato una soluzione ottima. Se invece il rilassamento risulta impossibile, allora anche il problema originario è impossibile, quindi l'algoritmo termina.

2. Altrimenti, se non si ricade nei casi precedenti, si identifica una separazione F^* di F, vale a dire un insieme finito F^* di sottoinsiemi di F, tale che:

$$\bigcup_{\forall F_i \in F^*} F_i = F \tag{1.5}$$

   Ogni sottoinsieme F_i è detto figlio di F. Tale fase, detta branching, è giustificata dal fatto che se F^* è una separazione, allora:

$$min\ \{\ c(x) \mid x \in F\} = min\ \{\ min\ \{\ c(x) \mid x \in F_i\} \mid F_i \in F\} \tag{1.6}$$

   F^* è spesso, anche se non necessariamente, una partizione di F. I sottoinsiemi di F vengono aggiunti alla coda dei sottoproblemi da processare.

3. Si seleziona un sottoproblema P_i dalla coda e se ne risolve un rilassamento. Ci sono quattro possibili casi:

   (a) Se si trova una soluzione ammissibile migliore dell'incumbent $\bar{x}$, allora si sostituisce $\bar{x}$ con la nuova soluzione e si continua.

(b) Se il sottoproblema non ammette soluzione, allora si scarta (pruning by infeasibility).

(c) Se troviamo una soluzione del rilassamento che ha costo maggiore o uguale a z, allora è possibile scartare il sottoproblema (pruning by optimality), in quanto non può portare ad una soluzione migliore di quella che si conosce già.

(d) Se non si verifica uno dei casi precedenti allora è necessario fare ulteriormente branching e aggiungere i figli del sottoproblema alla lista dei sottoproblemi da processare.

4. Si continua in questo modo fino a svuotare la lista dei sottoproblemi. Quando ciò avviene, l'incumbent corrente sarà la soluzione ottima del problema iniziale.

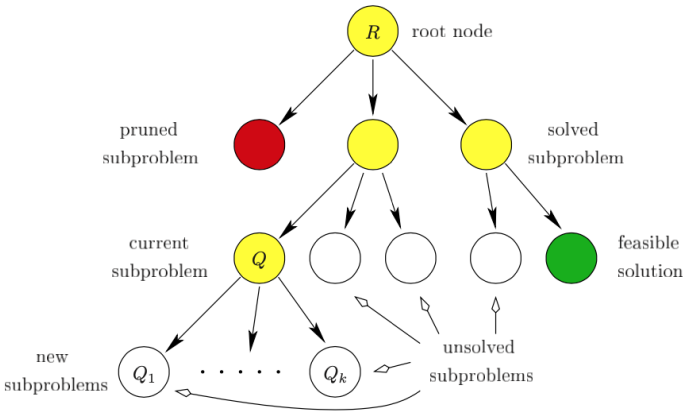
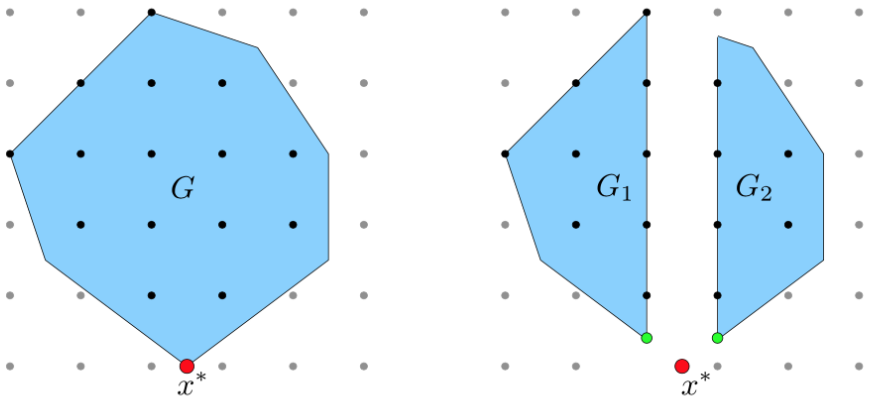


Figura 1.3: B&B

L'algoritmo B&B generico appena descritto si specializza per la risoluzione di problemi MIP, basta scegliere come calcolare rilassamenti e separazioni.
Per quanto riguarda il rilassamento, la scelta consiste nell'usare il rilassamento lineare dei sottoproblemi. Se la soluzione di tale rilassamento non è intera (il vincolo di interezza sulle variabili è l'unico rilassato), allora possiamo scegliere una x_j con un valore frazionario ${x_j}^*$ nella soluzione e considerare la seguente partizione:

$$x_j \leq \lfloor {x_j}^* \rfloor \vee x_j \geq \lceil {x_j}^* \rceil \tag{1.7}$$

Figura 1.4: Separation



1.3.2 Cutting Planes

L'algoritmo Cutting Planes (Piani di Taglio) si basa sul fatto di raggiungere, tramite raffinazioni successive, la formulazione ideale (Convex Hull) di un problema MIP. Quest'ultima consiste nell'avere un poliedro, definito dai vincoli del rilassamento lineare del problema, i cui vertici sono tutti interi, di conseguenza la soluzione ottima del rilassamento coinciderà con la soluzione ottima del problema MIP.
L'algoritmo segue questi step:

1. Dato il problema si risolve il suo rilassamento lineare, se si trova una soluzione frazionaria si continua, se invece la si trova intera l'algoritmo termina.

2. Si risolve il problema di separazione: data una soluzione frazionaria x^*, trovare una disugualianza valida $\alpha^T x \leq \alpha_0$, se esiste, violata da x^*, cioè tale che $\alpha^T x^* > \alpha_0$, quest'ultima porta alla definizione di un piano di taglio il quale verrà aggiunto come nuovo vincolo al problema.

3. Si ripete iterativamente fino ad arrivare alla formulazione ideale (Convex Hull).

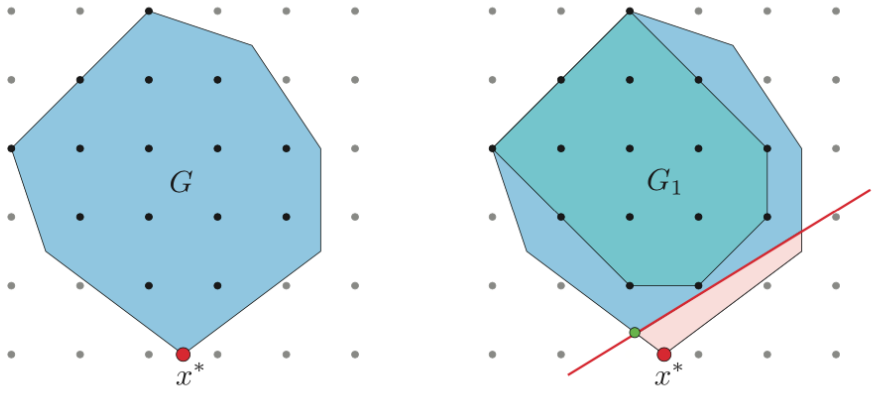


Figura 1.5: Cutting planes

Se si raggiunge la formulazione ideale allora si ha che la soluzione ottima del rilassamento lineare è intera, quindi è ottima anche per il problema originale.
La formulazione ideale esiste sempre ed è dimostrato che l'algoritmo converga. In pratica però, la formulazione ideale può avere un numero esponenziale di vincoli, inoltre è richiesto un algoritmo efficiente per la risoluzione del problema di separazione. Quindi in generale il Cutting Planes è peggiore del Branch & Bound (B&B).

1.3.3 Algoritmo Branch & Cut (B&C)

Una versione migliorata dell'algoritmo B&B, sviluppata specificatamente per il caso MIP, è nota come Branch & Cut (B&C). L'algoritmo B&C è essenzialmente un algoritmo ibrido che combina gli approcci di B&B (1.3.1) e Piani di Taglio (1.3.2), si basa sulla seguente idea: ad ogni nodo dell'albero decisionale, la formulazione associata al rilassamento del sottoproblema corrente può essere rafforzata mediante la generazione di piani di taglio.
Tale rafforzamento viene fatto per due motivi:

1. per ottenere una soluzione intera del rilassamento lineare
2. per ottenere un lower bound migliore e quindi più efficace per il pruning

L'approccio combinato permette di avere un algoritmo più performante rispetto al B&B puro e quindi rispetto anche il Cutting Planes puro. Nonostante l'idea

di base sia semplice, l'implementazione non è banale, difatti l'algoritmo ad ogni nodo eseguirà il Cutting Planes e successivamente il B&B; comincerà ad applicare quest'ultimo quando il trade-off tra piani di taglio e costo computazionale non sarà più vantaggioso.

1.4 Definizioni teoria dei grafi

Un grafo è una particolare struttura che consiste in un insieme di punti (vertici o nodi) e un insieme di linee (archi o rami) che uniscono coppie di nodi; formalmente è un insieme in cui è definita una relazione di qualunque tipo e, pertanto, la teoria dei grafi trova largo impiego nelle scienze.
Dato un grafo possiamo definire 2 sottoinsiemi:

1. Stable set cioè un insieme di nodi (o vertici) tali che non siano collegati da archi.

2. Vertex cover cioè un insieme di nodi (o vertici) complementari ad uno stable set.

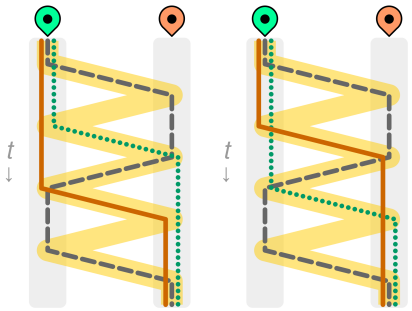
Si definisce inoltre il numero di vertex cover cioè la cardinalità del vertex cover più piccolo e lo si indica con $\tau(G)$.

1.5 Problema di Alcuin

Alcuino di York fu un pensatore e teorico anglossasone vissuto nell'VIII secolo dopo cristo. Alcuino si dedicò alla matematica con un puro interesse didattico, e affrontò molti problemi della cosidetta "matematica ricreativa", che si prefigge lo scopo di migliorare le abilità mentali dei giovani lettori attraverso quesiti di varia difficoltà. Il quesito più celebre elaborato da Alcuino è il problema dell'attraversamento del fiume ed è così formulato:

Molto tempo fa un contadino andò al mercato e comprò un lupo, una capra e un cesto di cavoli. Ritornando a casa, arrivò sulla riva di un fiume e noleggiò una barca per attraversarlo, ma la barca poteva trasportare(oltre a lui) soltanto uno tra il lupo, la capra e i cavoli. Se lasciati da soli senza la sua presenza, il lupo avrebbe mangiato la capra, oppure la capra avrebbe mangiato i cavoli; il lupo, essendo carnivoro, non avrebbe mangiato i cavoli. Il dilemma del contadino

è quindi il seguente: come li avrebbe potuti trasportare per intero sull'altra riva del fiume, evitando di lasciare incustoditi il lupo con la capra o la capra con i cavoli?



[15]

Figura 1.6: Problema di Alcuino

In questo elaborato si analizza una generalizzazione di questo problema che consiste nel considerare un grafo di input generico i cui nodi sono gli "elementi da trasportare" mentre gli archi identificano il legame di incompatibilità tra due elementi se lasciati dalla "stessa parte del fiume". Nel caso del problema proposto da Alcuino quindi, avremo questo grafo di input:

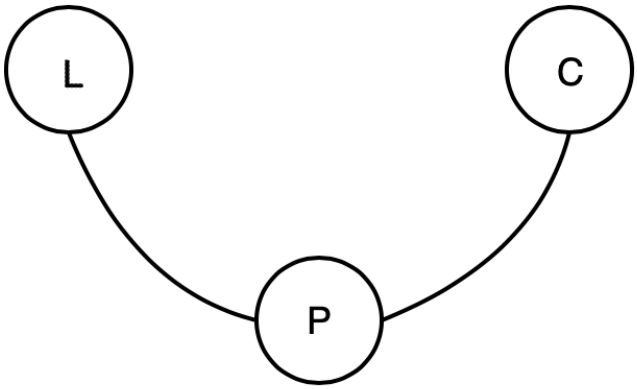


Figura 1.7: L: Lupo, P: Capra, C: Cavoli

La "capacità minima della barca" necessaria a rendere il problema risolvibile è chiamata numero di alcuin di un grafo (Alcuin(G)), ed è stato dimostrato nel

paper[10] che vale sempre:

$$\tau(G) \leq Alcuin(G) \leq \tau(G) + 1. \tag{1.8}$$

Nel paper[10] viene dimostrato inoltre, che il calcolo del numero di Alcuin di un grafo (Alcuin(G)) è NP hard, cioè non esistono algoritmi che lo risolvono in un tempo polinomiale. Infine viene dimostrato che "la barca" al massimo farà un numero di gite $n \leq 2|V| + 1$, dove V è l'insieme dei vertici (nodi) del grafo considerato. Dato questo risultato i modelli generati avranno un numero di fasi massimo pari a $2|V| + 1$, questo perchè eventuali fasi successive non verrebbero mai utilizzate.

Capitolo 2

Prove Sperimentali

2.1 Modello MIP

Per risolvere il problema è necessario scriverlo sottoforma di modello matematico, e successivamete tradurlo in codice Python[1]; in modo che il calcolatore possa risolverlo. Per scrivere il modello matematico si seguono 5 step:

1. Definizione degli insiemi
2. Identificazione parametri (eventualmente indicizzati sugli insiemi)
3. Definizione variabili (eventualmente indicizzate sugli insiemi)
4. Identificazione della funzione obiettivo
5. Identificazione dei vincoli

2.1.1 Insiemi

Gli insiemi necessari sono:

1. G = (V, E) rappresenta il grafo dove
   (a) E = insieme dei rami (archi) del grafo, in sorgente Python[1]:

   ```
   model.Edges = Set()
   ```

   (b) V = insieme dei vertici (nodi) del grafo, in sorgente Python[1]:

```
model.Nodes = Set()
```

2. L = {S,B,D} l'insieme dei luoghi possibili, in sorgente Python[1]:

   ```
   model.Places = Set()
   ```

3. F = $\{0, \ldots, 2|V|+1\}$ l'insieme delle possibili fasi, il limite superiore è dato dalla dimostrazione presente nel paper[10]. In sorgente python[1]:

   ```
   model.Trips = Set()
   ```

2.1.2 Paramateri

I parametri necessari sono:

1. K che rappresenta la capacità, in sorgente Python[1]:

   ```
   model.Capacity = Param()
   ```

2. $|V|$ che rappresenta il numero degli elementi del grafo, in sorgente Python[1]:

   ```
   model.len = Param()
   ```

2.1.3 Variabili

Sono necessarie 2 tipologie di variabili:

1. $$x_{ilf} = \begin{cases} 1 \; se\; il\; vertice\; i\; si\; trova\; nel\; luogo\; l\; durante\; la\; fase\; f \\ 0 \; altrimenti \end{cases} \tag{2.1}$$

   In sorgente Python[1]:

   ```
   model.x = Var(model.Nodes, model.Places,model.Trips,
      domain=Boolean)
   ```

2.
$$y_f = \begin{cases} 1\ se\ uso\ la\ fase\ f \\ 0\ altrimenti \end{cases} \tag{2.2}$$

In sorgente Python[1]:

```
model.y=Var(model.Trips, domain=Boolean)
```

2.1.4 Funzione obiettivo

Lo scopo del modello è minimizzare il numero di fasi quindi la funzione obiettivo rispecchia questo proposito.

$$min \sum_{\forall f \in F} y_f - 1 \tag{2.3}$$

In sorgente Python[1]:

```
def obj_rule(model):#f.obiettivo
   return sum(model.y[f] for f in model.Trips)-1
```

Nella formula il -1 serve perchè la fase 0 viene per forza usata, ma corrisponde alla fase iniziale quindi non bisogna includerla nel conteggio.

2.1.5 Vincoli

1. **Vincolo di partenza**
   Impone che tutti gli elementi del grafo si trovino a sinistra nella fase iniziale (fase 0).
   $$\sum_{\forall i \in V} x_{i's'0} = |V| \tag{2.4}$$

   In sorgente Python[1]:

   ```
   def constr_rule1(model):#vincolo di partenza
      return sum(model.x[i,"s",0] for i in
         model.Nodes)==model.len
   ```

2. **Vincolo di arrivo**
   Impone che tutti i nodi del grafo si trovino a destra o nella barca durante

l'ultima fase (fase $2|V|+1$).

$$\sum_{\forall i \in V} x_{i'd'(2|V|+1)} + \sum_{\forall i \in V} x_{i'b'(2|V|+1)} = |V| \tag{2.5}$$

In sorgente Python[1]:

```
def constr_rule2(model):#vincolo di arrivo
   return sum(model.x[i,"d",2*model.len+1] for i in
      model.Nodes)+sum(model.x[i,"b", 2*model.len+1]
      for i in model.Nodes)==model.len
```

3. **Vincolo di partenza y**
   Impone che la prima fase venga per forza usata.

$$y_0 = 1 \tag{2.6}$$

In sorgente Python[1]:

```
def constr_rule3(model):#vincolo di partenza y
   return model.y[0]==1
```

4. **Vincolo di un oggetto in un solo luogo**
   Impone che un nodo (vertice) si possa trovare solo in un luogo ad ogni fase.

$$\sum_{\forall l \in L} x_{ilf} == 1 \; \forall i \in V \; \forall f \in F \tag{2.7}$$

In sorgente Python[1]:

```
def constr_rule4(model, i, f):#vincolo un oggetto in
   un solo luogo
   return sum(model.x[i,l,f] for l in model.Places)==1
```

5. **Vincolo di progressione di y**
   Impone la regola di progressione per le fasi, questo serve al modello per capire che appena imposta una $y_f = 0$ tutte le successive dovranno essere impostate a 0. Invece se una fase viene impostata ad 1 ($y_f = 1$), tutte le precedenti devono essere impostate ad 1.

$$y_f \leq y_{f-1} \; \forall f \in F | \; f > 0 \tag{2.8}$$

In sorgente Python[1]:

```
def constr_rule5(model, f):#vincolo di progressione y
    if f>0:
        return model.y[f]<=model.y[f-1]
    else:
        return Constraint.Skip
```

6. **Vincolo legame y e x**
   Quando la fase non viene usata tutti i nodi (vertici) devono essere a destra, se invece ci sono ancora nodi (vertici) nella barca o a sinistra la fase deve essere utilizzata. Questo vincolo si traduce in 2 espressioni matematiche:

   (a)
   $$\sum_{\forall i \in V} x_{i's'f} + \sum_{\forall i \in V} x_{i'b'f} \geq y_f \; \forall f \in F \tag{2.9}$$

   In sorgente Python[1]:

   ```
   def constr_rule6(model,f): #vincolo legame y x 1
       return sum(model.x[i,"s",f] for i in
           model.Nodes)+sum(model.x[i,"b",f] for i in
           model.Nodes)>=model.y[f]
   ```

   (b)
   $$\sum_{\forall i \in V} x_{i's'f} + \sum_{\forall i \in V} x_{i'b'f} \leq |V| y_f \; \forall f \in F \tag{2.10}$$

   In sorgente Python[1]:

   ```
   def constr_rule7(model, f): #vincolo legame y x 2
       return sum(model.x[i,"s",f] for i in
           model.Nodes)+sum(model.x[i,"b",f] for i in
           model.Nodes)<=model.len*model.y[f]
   ```

7. **Vincolo di capacità**
   Impone che vengano trasportati un numero di nodi inferiore alla capacità della barca.
   $$\sum_{\forall i \in V} x_{i'b'f} \leq K y_f \; \forall f \in F \tag{2.11}$$

   In sorgente Python[1]:

```
def constr_rule8(model, f): #vincolo capacita'
   return sum(model.x[i,"b",f] for i in
      model.Nodes)<=model.Capacity*model.y[f]
```

8. **Vincoli gite**
   Impone la regola per le "gite" cioè se un elemento si trova sulla barca in una data fase, nella fase successiva si può trovare o nella barca o nella sponda destra o sinistra a seconda della gita, quindi in questo vincolo si distinguono 2 casi:

   (a) Gite pari: da destra a sinistra

   $$x_{i's'f} + x_{i'b'f} = x_{i's'f+1} + x_{i'b'f+1} \; \forall i \in V, \; \forall f \in F | \; f\%2 = 0 \quad (2.12)$$

   In sorgente Python[1]:

   ```
   def constr_rule9(model, i, f): #vincolo gite pari
      if f%2==0:
         return model.x[i,"s",f]+model.x[i, "b",
            f]==model.x[i,"s",f+1]+model.x[i, "b",
            f+1]
      else:
         return Constraint.Skip
   ```

   (b) Gite dispari: da sinistra a destra

   $$x_{i'd'f} + x_{i'b'f} = x_{i'd'f+1} + x_{i'b'f+1} \; \forall i \in V, \; \forall f \in F | \; f\%2 \neq 0 \quad (2.13)$$

   In sorgente Python[1]:

   ```
   def constr_rule10(model, i, f): #vincolo gite
      dispari
      if (f%2!=0) and (f<(2*model.len+1)):
         return model.x[i,"d",f]+model.x[i, "b",
            f]==model.x[i,"d",f+1]+model.x[i, "b",
            f+1]
      else:
         return Constraint.Skip
   ```

9. **Stable set a sinistra e destra**
   Impone che ad ogni fase ci sia uno stable set a sinistra e a destra eccetto per fase iniziale e fase finale.

   (a) Stable set a sinistra eccetto fase 0:

$$x_{i's'f} + x_{j's'f} \leq 1 \, \forall (i,j) \in E, \, \forall f \in F | f > 0 \tag{2.14}$$

   In sorgente Python[1]:

```
def constr_rule11(model, i, f): #stable set a
    sinistra
    #aggiunta
    l=i.split("-")
    #------
    if f>0:
        return
            model.x[int(l[0]),"s",f]+model.x[int(l[1]),
            "s", f]<=1
    else:
        return Constraint.Skip
```

   (b) Stable set a destra eccetto fase finale:

$$x_{i'd'f} + x_{j'd'f} \leq 2 - y_f \, \forall (i,j) \in E, \, \forall f \in F \tag{2.15}$$

   In sorgente Python[1]:

```
def constr_rule12(model, i, f): #stable set a
    destra
    #aggiunta
    l=i.split("-")
    #------
    return
        model.x[int(l[0]),"d",f]+model.x[int(l[1]),
        "d", f]<=2-model.y[f]
```

2.2 Descrizione prove sperimentali

I calcoli sono stati eseguiti sul cluster del dipartimento dedicato alla ricerca operativa, nello specifico sono state usate le lame arrow che sono equipaggiate con:

1. Intel(R) Xeon(R) CPU E5-2623 v3 @ 3.00GHz
2. 16 GB di ram

Per risolvere i modelli è stato usato CPLEX 22.1.1[9] e la sua interfaccia su Python[1] tramite il pacchetto Pyomo[3]. Inotre è stato necessario l'uso dello scheduler SLURM[8] per l'utilizzo del cluster del dipartimento.
I modelli considerati usano grafi con 100 vertici, questa scelta è dovuta al fatto che grafi più piccoli risultavano troppo semplici e grafi più grandi invece generavano un modello troppo grande per rispettare i vincoli di memoria del calcolatore (CPLEX [9] running out of memory error).

2.2.1 Generazione grafi casuali

La generazione dei grafi viene effettuata tramite il pacchetto python Networkx[2] utilizzando la funzione *gnprandomgraph(n,p,seed)*, i cui parametri rispettivamente indicano: il numero di nodi (vertici) del grafo, la probabilità di generazione degli archi (rami), il seed per avere un risultato random ma riproducibile. I grafi ottenuti vengono salvati su un file tramite la funzione *write_adjlist(graph,filename)* dove i parametri indicano rispettivamente il grafo e il filename dove viene scritta l'adjacency list (lista di adiacenze) del grafo. Quest'ultima usa una rappresentazione del nodo sottoforma di stringa la quale segue questa struttura:

NODO [VICINO] [VICINO] ... [VICINO]

Inoltre viene usata la funzione *approximation.maximum_independent_set(graph)* del pacchetto Networkx[2] per ottenere, tramite approssimazione lo stable set più grande, dove il parametro è il grafo stesso. Questa funzionalità viene usata per poi calcolare la capacità della barca. Difatti come dimostrato nel paper[10] il numero di alcuin di un grafo (Alcuin(G)) è limitato dal numero di vertex cover $\tau(G)$ vedi (1.8). Quindi la capacità che corrisponde al numero di Alcuin è calcolata come la differenza tra la quantità di nodi del grafo e la quantità di vertici dello stable set più grande, questa differenza ci da la quantità di nodi del vertex cover più piccolo, cioè il numero di vertex cover $\tau(G)$, in sorgente python:

```
Capacity=len(G1)-len(nx.approximation.maximum_independent_set(G1))
```

Essendo un'approssimazione il risultato del calcolo si può diminuire ulteriormente così da rendere il problema più complesso, anche se in alcuni casi questo porta il modello ad essere infeasibile.

2.2.2 Conversione per modello

Un modello espresso tramite Python[1] e Pyomo[3] può essere un concrete model oppure un abstract model. La differenza tra i due è che il concrete model prevede che il codice definisca insiemi e parametri tramite tipologie di dati Python[1], quindi lo si può vedere come un modello "hard-coded", cioè il codice può costruire solo e soltanto quel modello a meno di modifiche dei dati nel codice stesso. Invece un abstract model permette di definire insiemi e parametri in maniera generica e al momento dell'esecuzione deve essere dato in input un file .dat che permette di specificare i dati del modello. Questo permette di scrivere il codice una volta sola e di riusarlo per tutti i file .dat compatibili. Nel caso di studio è stato scelto di usare un abstract model, quindi c'è stata la necessità di generare i file .dat di ogni grafo a partire dalle adjancency list. Per far questo è stato scritto uno script Python[1] che legge il file dell'adjancency list e scrive il corrispondente file .dat seguendo questa struttura:

set Places := s b d ;
set Nodes := ... ;
set Edges:= .... ;
param Capacity :=XX;
param len := 100;

Per il valore del parametro Capacity viene aggiunto nel file della lista di adiacenze una riga aggiuntiva identificata da un carattere speciale, in cui viene scritto il risultato di 2.2.1.

2.2.3 Generazione file SLURM

Per poter eseguire programmi nel cluster del dipartimento è stato necessario generare file per lo scheduler generico SLURM[8] il quale gestisce una coda di job e sceglie quale eseguire in base alla priorità e alla disponibilità di risorse. Per inserire nella coda nuovi job è necessario fare il sumbit tramite il comando *sbatch –wckey=rop –requeue nomefile*.

Per ogni instanza quindi si genera il suo corrispondente file SLURM[8] tramite uno script Python[1]. I vari file generati seguono questa struttura generale:

```
#!/bin/bash
#SBATCH --jobname=ModelloAlcuin_data_GraphXXX
#SBATCH --partition=arrow
#SBATCH --ntasks=1
#SBATCH --mem=14GB
#SBATCH --output output_%j.txt
#SBATCH --error errors_%j.txt
sudo cpupower frequency-set -g performance
sleep 0.1
stress-ng -c 4 --cpu-ops=100
ulimit -v 16777216
. ~/.bashrc
cd /home/sprocatti1/AlcuinProblem
python3 Model.py Generatore\
   grafi/ConvertitiCasuali/XXX.dat X >Output\
   terminale/Output_XXX.txt
sudo cpupower frequency-set -g powersave
```

Le righe che iniziano con '#SBATCH' sono direttive per lo scheduler. In particolare:

1. *#SBATCH –jobname=ModelloAlcuin_data_GraphXXX* imposta il nome del job
2. *#SBATCH –partition=arrow* seleziona la partizione del cluster in cui eseguire i job, in questo caso arrow
3. *#SBATCH –ntasks=1* imposta l'esclusività dei run
4. *#SBATCH –mem=14GB* imposta la memoria massima utilizzabile
5. *#SBATCH –output output_%j.txt* imposta il file di output del job, %j verrà sostituito con l'id del job.
6. *#SBATCH –error errors_%j.txt* imposta il file di errore del job, %j verrà sostituito con l'id del job.

Dopo le direttive si porta il processore in modalità performance, si aspettano 0.1 s e poi si fa uno stress test. Si caricano quindi, alcune variabili d'ambiente impostate

nel file .baschrc e solo successivamente si eseguono i comandi per l'esecuzione. Questi comprendono il cambio di working directory e il comando per l'esecuzione del sorgente Python[1], il quale prevede 2 parametri:

1. il file .dat che descrive i parametri e gli insiemi del modello,
2. il seed del risolutore.

Finita l'esecuzione si riporta il processore in modalità powersave.

Capitolo 3

Analisi risultati

I parametri dei risultati di ogni esecuzione vengono salvati in un unico file .csv (*Comma-Separated Values*), questa scelta è stata fatta perchè i file .csv sono molto usati in ambito di elaborazione dati e sono compatibili con tanti programmi software. È stato impostato un file di output diverso per ogni modello poichè le lame del cluster eseguono questi ultimi in parallelo e hanno anche home condivisa. Alla fine della risoluzione dei diversi modelli, i vari file vengono combinati assieme tramite uno script Python[1]. Il file .csv ha la seguente struttura:

GRAPH	TIME	LB	UB	TERM	CAP	LENGTH	PROB*100	MINCAP
data_Graph40_1_41	41.71	3.0	3.0	optimal	90	100	40	1
data_Graph90_1_17	402.05	None	None	infeasible	95	100	90	1

Per generare dati sufficienti per un'analisi statistica sono stati variati i seguenti parametri:

1. La probabilità degli archi: [0.4, 0.5, 0.6, 0.7, 0.8, 0.9]
2. Il seed di generazione del grafo
3. La diminuzione di capacità rispetto al risultato dell'espressione 2.2.1: [0, 1, 2]
4. Il seed del risolutore che influenza le decisioni durante il processo di risoluzione

I dati salvati nel file .csv sono stati poi elaborati tramite uno script Python[1] per presentare i risultati in maniera grafica.

3.1 Confronto tra le varie probabilità

Questo confronto viene fatto per trovare eventualmente un legame tra i tempi di calcolo e la densità del grafo. Per far questo sono stati elaborati dei grafici in cui si confrontano i tempi di calcolo al pari del risultato finale (ottimo o infeasible) e al pari della diminuzione di capacità.

3.1.1 Ottimi

Partendo dalla considerazione dei tempi di calcolo riguardanti problemi risolti all'ottimo si può notare dai tre grafici come non ci sia un pattern regolare nonostante il numero di istanze analizzate quindi, si può osservare come il tempo non sia monotono rispetto alla densità del grafo, infatti quest'ultima è solo uno dei parametri che influenza il tempo di calcolo. L'andamento irregolare è molto visibile in figura 3.2 dove ci sono vari "picchi" senza una crescita o decrescita costante. In figura 3.3, invece, si può notare che grafi molto densi risultano tutti infeasible difatti il grafico presenta punti solo fino a probabilità dei rami uguale a 0.7, questo suggerisce che l'approssimazione della libreria potrebbe essere molto precisa per grafi densi.

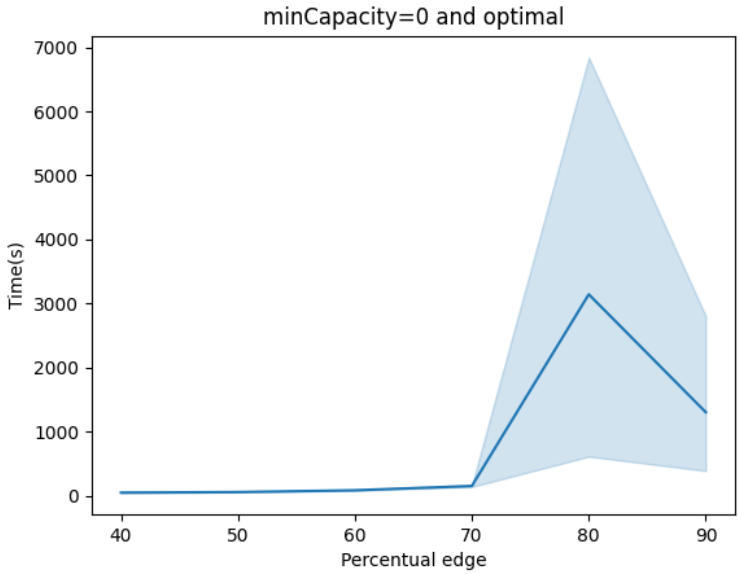


Figura 3.1: minCapacity=0 and optimal

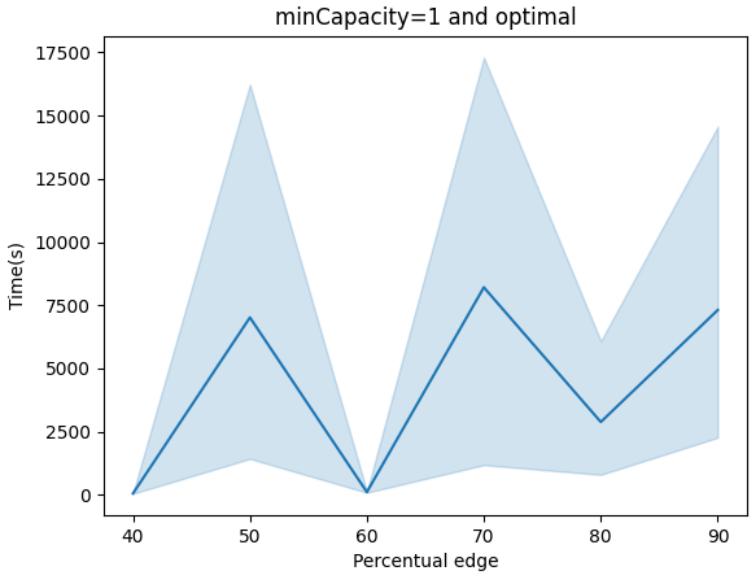


Figura 3.2: minCapacity=1 and optimal

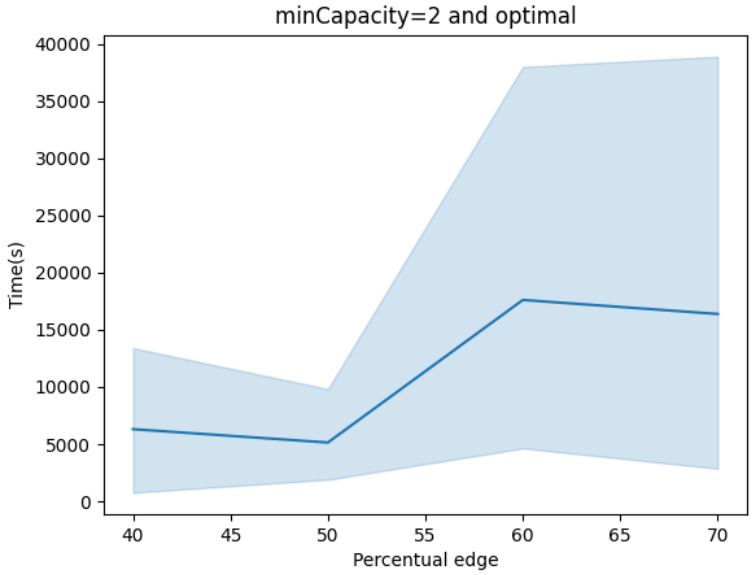


Figura 3.3: minCapacity=2 and optimal

3.1.2 Infeasible

L'osservazione fatta per il caso dei problemi risolti all'ottimo è valida anche nel caso dei problemi infeasible, infatti anche in questi ultimi non c'è un pattern regolare. Tra i grafici qui presentati non c'è il grafico relativo alla diminuzione di capacità uguale a 0 poichè esso non presenta punti, difatti questi ultimi sono risultati tutti risolvibili all'ottimo.
Osservando la figura 3.4 si può vedere come solo grafi densi risultino infeasible con diminuzione uguale ad 1, questo suggerisce che l'approssimazione della libreria sia molto buona per grafi con probabilità degli archi elevate.

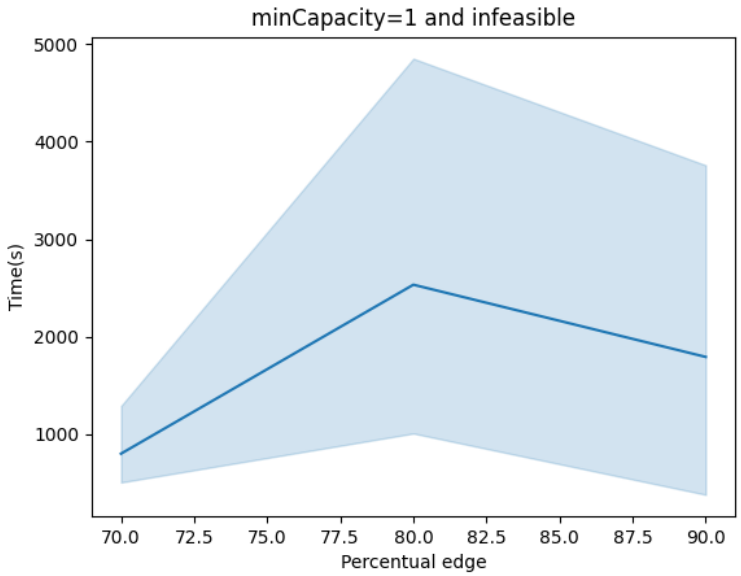


Figura 3.4: minCapacity=1 and infeasible

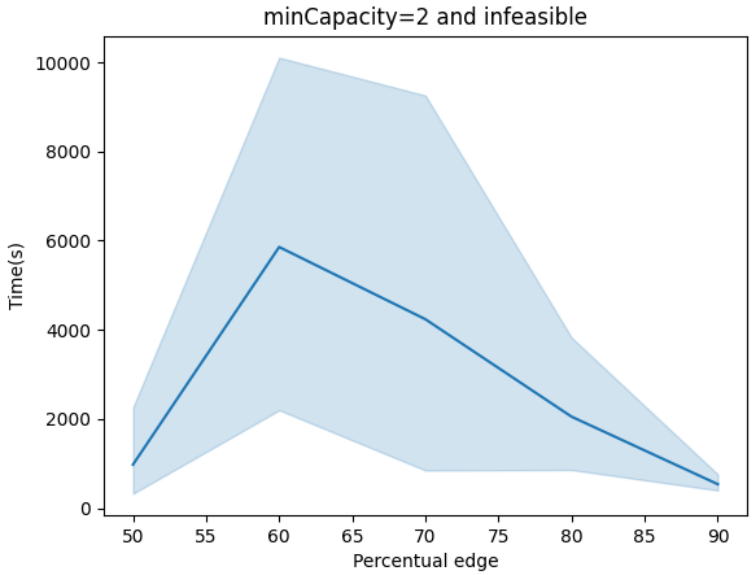


Figura 3.5: minCapacity=2 and infeasible

3.2 Confronto tra le varie capacità

Questo confronto viene fatto per capire come variano i tempi di calcolo al diminuire della capacità della barca. Per capire l'eventuale legame sono state considerate tutte le probabilità ma vengono distinti i casi in base al risultato finale dell'elaborazione (ottimo o infeasible).

3.2.1 Ottimi

Nel caso di problemi risolti all'ottimo si può notare dal grafico che diminuendo la capacità il problema risulta via via più complesso indipendentemente dalla probabilità degli archi. Difatti il subset di vertici da "trasportare" è sempre più piccolo e di conseguenza gli stable set che permettono di soddisfare i vincoli devono essere più grandi, quindi il computer impiega più tempo a trovare la giusta sequenza per il "trasporto".

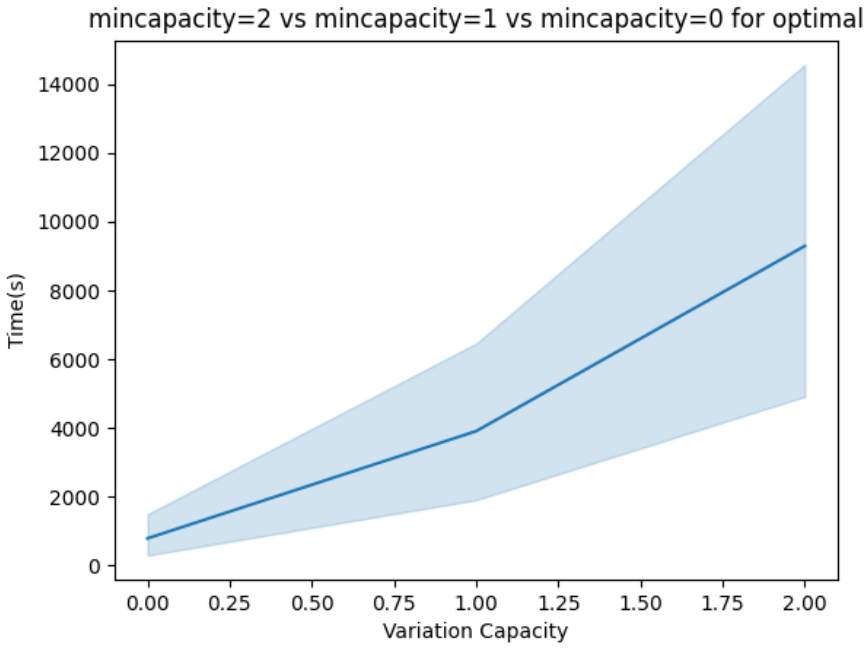


Figura 3.6: mincapacity=2 vs mincapacity=1 vs mincapacity=0 for optimal

3.2.2 Infeasible

Se invece si considera il caso di problemi infeasible si può notare che nessun problema con diminuzione di capacità uguale a 0 risulta infeasible, come detto precedentemente. Nel grafico quindi, saranno presenti solo due valori di diminuzione di capacità e osservandolo si può vedere come ci sia una leggera crescita dei tempi all'aumentare della diminuzione, questo potrebbe essere dovuto al fatto che il computer ha più "combinazioni possibili" da provare per dire con assoluta certezza che il problema risulta infeasible.

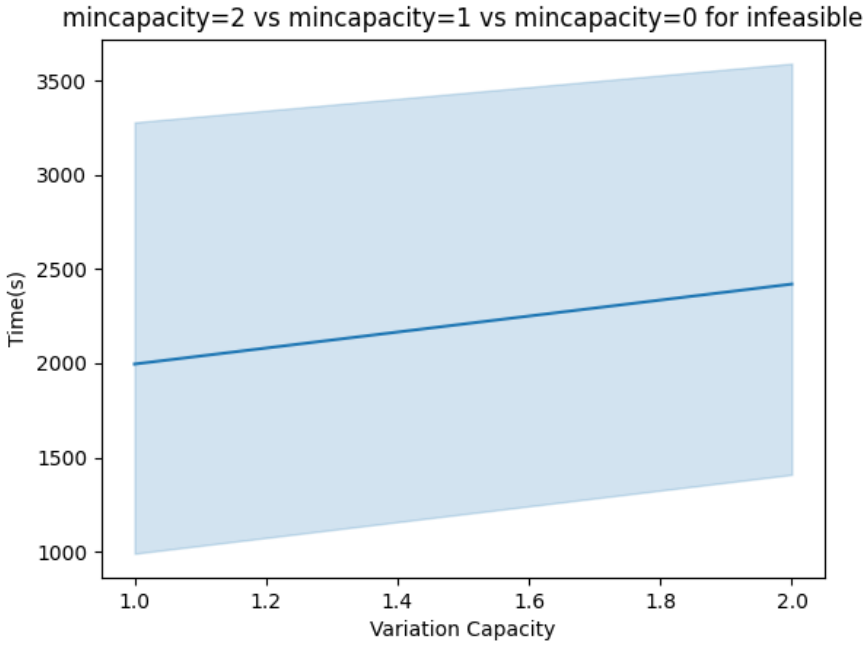
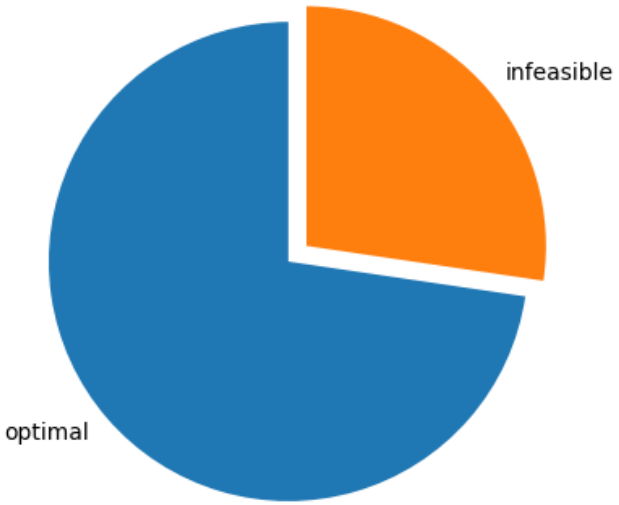


Figura 3.7: mincapacity=2 vs mincapacity=1 vs mincapacity=0 for infeasible

3.3 Ottimi e infeasible

Da questo grafico possiamo notare come tanti problemi considerati risultano risolvibili all'ottimo, questo deriva dal fatto che tutti i grafi considerati che hanno diminuzione di capacità pari a 0 ammettono soluzione ottima, come evidenziato precedentemente; di conseguenza il grafico a torta presenta uno sbilanciamento verso questa parte. La "fetta" riguardo ai problemi infeasible è composta principalmente da grafi molto densi con diminuzione uguale a 2, difatti questi ultimi risultano tutti infeasible come si può vedere nella figura 3.3.

Figura 3.8: Grafico a torta



3.4 Numero di Alcuin

Per fare un'analisi corretta della relazione tra numero di Alcuin e densità del grafo sono stati considerati i problemi risultati infeasible per una data diminuzione, allora il numero di Alcuin di questi grafi è la capacità aumentata di 1 (se un grafo risulta infeasible con diminuzione uguale ad 1 allora non bisogna considerare l'entry del file .csv relativa allo stesso grafo ma con diminuzione uguale a 2). Da questa analisi risulta che il numero medio di Alcuin è uguale a 94.16, il che è molto alto rapportato alla dimensione del grafo (100). Inoltre come evidenziato dalla figura si ha che il numero di Alcuin cresce in maniera proporzionale alla densità del grafo, questa osservazione deriva dal fatto che più il grafo è denso più lo stable set massimo sarà piccolo, quindi il vertex cover minimo sarà più grande e di conseguenza anche il numero di vertex cover, il quale limita il numero di Alcuin come evidenziato nella formula 1.8.

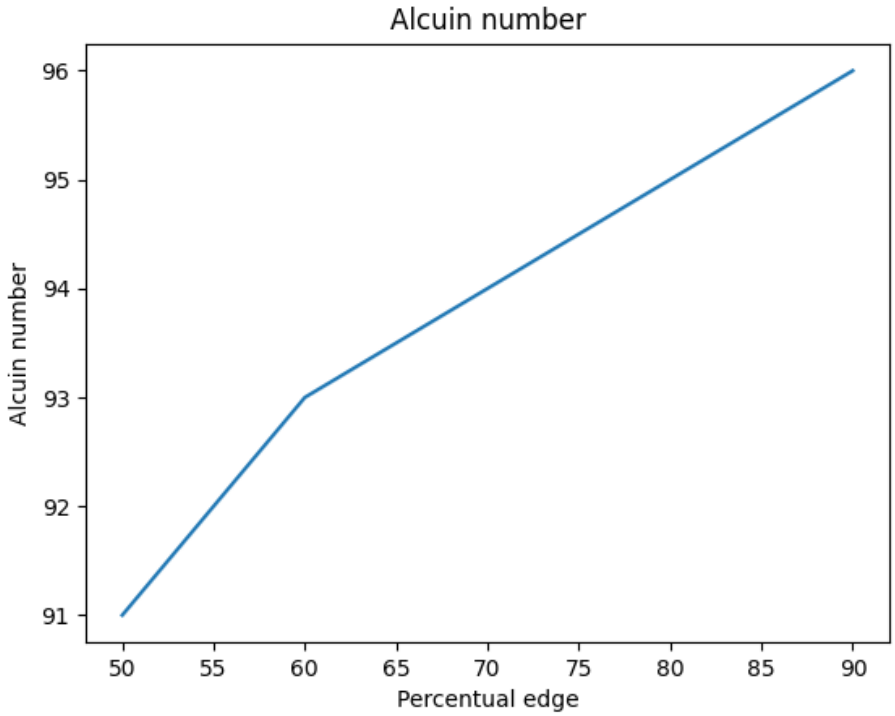


Figura 3.9: Numero di Alcuin

Capitolo 4

Conclusioni

La generalizzazione del problema del numero di Alcuin di un grafo affrontato in questo studio è un problema molto complesso, dipende da molti fattori come la struttura del grafo stesso, la sua sparsità/densità e l'approssimazione che si usa per calcolare il numero di Alcuin di partenza. Sopratutto quest'ultimo punto influenza molto lo studio. Nel caso di questo elaborato si è verificato che l'approssimazione riguardo allo stable set più grande fatta da Networkx[2] è abbastanza precisa per grafi molto densi infatti se vengono generati con probabilità degli archi uguale a 0.8 o 0.9 risultano tutti infeasible con diminuzione uguale a 2, mentre alcuni anche con diminuzione uguale ad 1. Grafi più sparsi risultano ancora risolvibili all'ottimo con diminuzione 2, quindi probabilmente l'approssimazione è meno precisa e si possono considerare diminuzioni ulteriori. Per vedere quanto buona è l'approssimazione si potrebbe provare con grafi abbastanza sparsi a diminuire ulteriormente la capacità fino a quando i problemi risultano risolvibili all'ottimo. Se non ci fosse stata questa funzionalità nella libreria[2] allora lo studio sarebbe diventanto molto più complesso poichè bisognava trovare questo numero che approssimava la dimensione del vertex cover più piccolo per ogni grafo.
Dati i grafici riportati nel capitolo precedente si può concludere che non c'è un legame di crescita o decrescita dei tempi di calcolo al variare della densità del grafo, al contempo si può notare come la diminuzione del possibile numero di Alcuin testato renda il problema via via più complesso come evidenziato in figura 3.6. Sempre dai grafici si può osservare un'alta variabilità dei tempi (fascia azzurrina), questa osservazione aggiunta, nel caso dei confronti tra probabilità, alla non regolarità delle curve tracciate nei grafici porta alla conclusione che ci siano altri parametri, oltre alla densità, che influenzano il tempo di calcolo.
In conclusione si può osservare come il numero di Alcuin di un grafo sia abba-

stanza alto rapportandolo alla dimensione del grafo stesso. Questa osservazione deriva dalla figura 3.9 in cui si può vedere che il numero di Alcuin varia tra 91 e 96 in grafi di dimensione 100, mentre il numero di Alcuin medio è 94.16. Inoltre anche nei grafi non considerati in quest'ultima stima si hanno numeri di capacità della barca molto elevati, quindi anche se non si è determinato il numero di Alcuin preciso per questi ultimi si può facilmente dedurre che non è molto distante dall'approssimazione di partenza essendo quest'ultima abbastanza precisa in generale.

Bibliografia

[1] Python Software Foundation. Python Language Reference, versione 3.9. Disponibile su https://docs.python.org/3/

[2] Networkx python package. Versione 3.1. Disponibile su https://networkx.org

[3] Pyomo python package. Versione 6.5.0. Disponibile su: http://www.pyomo.org/documentation

[4] MatPlotLib python package. Versione 3.7.1. Disponibile su: Hompage https://matplotlib.org/

[5] Seaborn python package. Versione 0.12.2. Disponibile su: https://seaborn.pydata.org

[6] Numpy python package. Versione 1.24.2. Disponibile su: https://numpy.org

[7] Pandas python package. Versione 2.0.1. Disponibile su: https://pandas.pydata.org

[8] SLURM scheduler generico. Docs disponibili su https://clusterdeiguide.readthedocs.io/en/latest/

[9] IBM, ILOG CPLEX Optimization Studio 22.1.1. https://www.ibm.com/docs/en/icos/22.1.1.

[10] Csorba, P., Hurkens, C. A. J.,& Woeginger, G. J. (2012). The Alcuin number of a graph and its connections to the vertex cover number. SIAM Review, 54(1), 141-154. https://doi.org/10.1137/110848840

[11] Domenico Salvagnin. Introduzione all'ottimizzazione discreta. Università degli Studi di Padova, 2018. Disponibile su: http://www.dei.unipd.it/ salvagni/didattica/intro.pdf

[12] Domenico Salvagnin. Cenni di programmazione lineare. Università degli Studi di Padova,2011. Disponibile su: http://www.dei.unipd.it/ salvagni/didattica/lp.pdf

[13] Domenico Salvagnin. Cenni di Programmazione Lineare Intera. Università degli Studi di Padova, 2020. Disponibile su: http://www.dei.unipd.it/ salvagni/didattica/mip

[14] Michele Sprocatti https://github.com/Sproc01/AlcuinProblem.git

[15] By Cmglee - Own work, CC BY-SA 4.0, https://commons.wikimedia.org/w/index.php?curid=112728719