Progetto AMOD

Fabiano Veglianti

Indice

1 Introduzione

Questo documento descrive l'implementazione dell'algoritmo Column Generation per la risoluzione del problema di Cutting Stock Monodimensionale.

1.1 Descrizione del problema

Nel problema di Cutting Stock (CSP) Monodimensionale si hanno a disposizione dei *Paper Roll* di lunghezza L e sono definiti dei *Moduli* di lunghezze diverse. Per ciascun modulo è definita una *Domanda* da soddisfare. Le domande sono soddisfatte andando a tagliare i Paper Roll in modo da ottenere i moduli delle lunghezze desiderate. Dunque, il problema di Cutting Stock Monodimensionale consiste nel *minimizzare* il numero di Paper Rolls tagliati per soddisfare le domande relative a tutti i moduli. È evidente che esistono molti modi di tagliare un Paper Roll in modo da ottenere i moduli, ciascun modo di tagliare il Paper Roll per ottenere i moduli prende il nome di *Cutting Pattern* e definisce quanti moduli delle varie lunghezze sono ottenute tagliando un Paper Roll.

2 Column Generation per CSP

Diamo una definizione formale del problema di Cutting Stock e dell'algoritmo Column Generation usato per risolvero.

2.1 Definizione formale del problema

I dati del problema sono i seguenti:

- L lunghezza del paper roll;
- l_i $i = 1, ..., m$ lunghezza del modulo i-esimo;
- d_i $i = 1, ..., m$ domanda relativa al modulo di lunghezza l_i.

NOTA: nel modello considerato ciascun modulo è identificato solo dalla propria lunghezza; dunque, due moduli diversi che abbiano la stessa lunghezza possono essere sostituiti senza perdita di generalità da un unico modulo che ha la stessa lunghezza dei moduli di partenza e la cui domanda è pari alla somma delle domande relative ai moduli di partenza.

Un cutting pattern è formalizzato attraverso un vettore $A^j \in \mathbb{Z}^m$:

$$A^j = \begin{bmatrix} a_{1j} \\ a_{2j} \\ \vdots \\ a_{mj} \end{bmatrix} \in \mathbb{Z}^m_{\geq 0}$$

ciascun elemento a_{ij} sta ad indicare quanti moduli di lunghezza l_i sono prodotti dal cutting pattern A^j.
Non tutti i cutting pattern sono ammissibili; infatti, affinché un cutting pattern sia ammissibile è necessario che: la somma delle lunghezze dei moduli generati dal cutting pattern non superi la lunghezza dei paper rolls e che, ovviamente, ciascun elemento a_{ij} sia un intero maggiore o uguale di 0.
Formalmente per il cutting pattern A^j:

$$\sum_{i=1}^{m} a_{ij} * l_i \leq L$$
$$a_{ij} \geq 0,\ a_{ij} \text{ intero} \forall i = 1, ..., m$$

Si noti che ogni volta che un cutting pattern viene applicato, un paper roll viene tagliato, dunque il problema può essere riformulato dicendo di voler minimizzare il numero di volte in cui i cutting pattern vengono applicati (contando più volte lo stesso cutting pattern, se questo viene applicato più volte).
Da questa osservazione possiamo definire le variabili del probelma:

$$x_j = \text{numero di volte in cui il cutting pattern } A^j \text{ viene usato}$$

Detto $T = \{j : A^j \text{ è un cutting pattern ammissibile}\}$, il numero delle variabili del problema è uguale al numero dei cutting pattern ammissibili, ma questi sono in numero troppi per essere considerati tutti. L'algoritmo Column Generation è pensato per quei problemi, come il Cutting Stock Monodimensionale, per cui è impossibile considerare esplicitamente tutte le variabili. Torneremo sull'algoritmo Column Generation tra poco.

In base a quanto detto possiamo formulare il problema di cutting stock monodimensionale come problema di programmazione lineare intera come segue:

$$z^I = \min_x \sum_{j \in T} x_j$$
$$\sum_{j \in T} a_{ij} * x_j \geq d_i \ i = 1, ..., m$$
$$x_j \in \mathbb{Z}_{\geq 0} \ \forall j \in T$$

con z^I valore ottimo della soluzione obiettivo e x^{*^I} soluzione ottima.

2.2 Rilassamento Lineare e Round-Up

Considerando il rilassamento lineare del problema, cioè

$$\begin{aligned} z^C = & \min_x \sum_{j \in T} x_j \\ & \sum_{j \in T} a_{ij} * x_j \geq d_i \ i = 1, ..., m \\ & x_j \in \mathbb{R}_{\geq 0} \ \forall j \in T \end{aligned}$$

se i valori delle variabili ottime del rilassamento lineare $x_j^{*^C}$ sono alte rispetto al numero di vincoli del problema m, allora l'errore compiuto considerando al posto di x^{*^I} il round-up della soluzione ottima del rilassamento lineare, cioè $\left\lceil x^{*^C} \right\rceil$, è piccolo.

Infatti, le uniche componenti di x^{*^C} maggiori di 0 sono le variabili in base, quindi sono al più m e possiamo supporre senza perdità di generalità che siano $x_1^{*^C}$,...,$x_m^{*^C}$.
Allora, dal momento che vale:

$$\left\lceil x_j^{*^C} \right\rceil \leq x_j^{*^C} + 1$$

definendo

$$z^{ROUND-UP} = \sum_{j=1}^{m} \left\lceil x_j^{*^C} \right\rceil$$

si ha:

$$\begin{aligned} & z^C \leq z^I \leq z^{ROUND-UP} \leq z^C + m \\ \text{errore assoluto} = & \ z^{ROUND-UP} - z^I \leq z^{ROUND-UP} - z^C \leq m \\ \text{errore relativo} = & \ \frac{z^{ROUND-UP} - z^I}{z^{z^{ROUND-UP}}} \leq \frac{m}{z^{ROUND-UP}} \end{aligned}$$

Da questo punto in poi consideriamo il rilassamento lineare del CSP.

2.3 Column Generation

2.3.1 Problema Ristretto

Iniziamo col considerare una versione ristretta del rilassamento lineare del CSP in cui consideriamo solo un sottoinsieme dei cutting pattern ammissibili: dunque, definito un insieme di indici $R \subseteq T$, si considerano solo i cutting pattern A^j con $j \in R$, conseguentemente il numero delle variabili del problema si riduce a $|R|$. La scelta **iniziale** di R deve soddisfare due requisiti:

- $|R|$ deve essere molto minore di $|T|$, cioè il numero delle variabili del problema ristretto deve essere trattabili.

- tramite il sottoinsieme di cutting pattern definiti da R deve essere possibile soddisfare tutte le domande.

Il problema ristretto è quindi:

$$\begin{aligned} &\min_{x_R} \mathbb{1}^\intercal x_R \\ &A_R * x_R \geq d \\ &x_R \geq 0 \end{aligned}$$

con x_R^* il suo punto di minimo. Si noti che x_R^* è un vettore in $\mathbb{R}_{\geq 0}^{|R|}$, ma può essere esteso ad un vettore in $\mathbb{R}_{\geq 0}^{|T|}$ ponendo a 0 quelle componenti relativi a indici in T che non sono in R.
Chiamiamo x il vettore così generato.
Il vettore x sarà una soluzione ammissibile per il rilassamento lineare del CSP perché le sue componenti diverse da 0 le abbiamo ricavate risolvendo il problema ristretto e dunque utilizzando i cutting pattern del problema ristretto, in numero come specificato dalle componenti diverse da 0, siamo in grado di soddisfare la domanda; tuttavia, in generale non sarà una soluzione ottima perché ponendo a 0 le componenti relativi a indici in T che non sono in R stiamo imponendo di non utilizzare altri cutting pattern al di fuori di quelli considerati nel problema ristretto.

2.3.2 Oracolo di Generazione delle Colonne

Ci chiediamo quindi se esiste un cutting pattern A^j non considerato in A_R e che invece sarebbe utile per diminuire la funzione obiettivo, cioè equivalentemente se esiste una variabile x_j il cui costo ridotto $\gamma_j < 0$.
Si noti che il vettore dei costi ridotti ha tante componenti quante sono le variabili del problema originale, quindi troppe per essere esplorate tutte esplicitamente.
Dalla dualità per la programmazione lineare si ha che:

$$\gamma_j = c_j - y^{*\intercal} A^j$$

con:

- c_j peso della variabile primali x_j nella funzione obiettivo, quindi nel caso di CSP $c_j = 1$;
- y^* soluzione ottima del duale **del problema ristretto**.

quindi la condizione di esistenza di una variabile x_j il cui costo ridotto $\gamma_j < 0$ diventa:

$$\gamma_j = 1 - y^{*\intercal} A^j < 0 \iff y^{*\intercal} A^j > 1$$

Il duale del problema ristretto è:

$$max\{y^\intercal d, y^\intercal A^R \leq c_R^\intercal, y \geq 0^m\}$$

mentre il duale del rilassamento lineare del problema originale è:

$$max\{y^\intercal d, y^\intercal A \leq c^\intercal, y \geq 0^m\}$$

con ciò, segue che la condizione $\gamma_j < 0 \iff y^{*\intercal} A^j > 1$ corrisponde all'esistenza nel duale di un vincolo violato dalla soluzione ottima del duale del problema ristretto e tale vincolo duale corrisponde alla variabile primale che deve entrare in base.

L'oracolo di generazione delle colonne A^j procede come segue: dopo aver risolto il problema ristretto, si ottiene per dualità forte la soluzione ottima del suo duale, cioè y^*, quindi è necessario verificare se esiste ed eventualmente individuare un vettore

$$A^j = \begin{bmatrix} a_1 \\ a_2 \\ \vdots \\ a_m \end{bmatrix}$$

per cui $y^{*\intercal} A^j > 1$ e che possa essere un cutting pattern ammissibile cioè soddisfi:

- $\sum_{i=1}^{m} a_i * l_i \leq L$
- $a_i \geq 0$, a_i intero$\forall i = 1, ..., m$

Per risolvere entrambe le questioni contemporaneamente è sufficiente considerare il seguente problema detto *problema di PRICING*:

$$\begin{aligned} \xi = &\max_a \sum_{i=1}^{m} y_i^{*R} * a_i \\ &\sum_{i=1}^{m} l_i * a_i \leq L \\ &a_i \in \mathbb{Z}_{\geq 0} \ \forall i = 1, ..., m \end{aligned}$$

dunque, possono accadere due cose:

- $\xi \leq 1 \implies \nexists$ cutting pattern A^j s.t. $y^{*\intercal} A^j > 1$;
  $\implies$ l'estenzione di x_R^* è una soluzione ottima per il rilassamento lineare del CSP originale;
- $\xi > 1 \implies$ la soluzione ottima del problema di PRICING a^* è un cutting pattern ammissibile che corrisponde ad una variabile x_j con $\gamma_j < 0$;
  $\implies$ bisogna aggiungere la colonna $A^j = a^*$ in A_R e reiterare la procedura.

3 Implementazione

L'implementazione dell'algoritmo di generazione delle colonne per il problema di cutting stock monodimensionale è stata realizzata in Python, utilizzando l'API di Gurobi per risolvere i problemi di ottimizzazione previsti: in particolare è stata usata la versione 10.0.0 di Gurobi e la versione 3.8 di Python.

3.1 Implementazione dell'algoritmo

Il file **esecuzioneSingola.py** continiene tutto il necessario per risolvere un'istanza di un problema di cutting stock monodimensionale.
Un'istanza del problema è descritta tramite 3 parametri:

- L: *intero* che rappresenta la lunghezza dei paper rolls;
- listOfModules: *list* (struttura dati di python) di *interi* che rappresentano le lunghezze $l_1, ..., l_m$ dei moduli;
- listOfDemands: *list* di *interi* che rappresentano le domande relative ai moduli corrispondenti.

In particolare, il metodo **computeSolution** in **esecuzioneSingola.py** è quello che risolve un'istanza del problema. Il metodo è composto dai seguenti passi:

1. Determinare i cutting patterns iniziali.
2. Iterare:

2.1. Risolvere il rilassamento lineare del problema ristretto.

2.2. Risolvere il problema di Pricing.

2.3. Uscire dal ciclo se il valore ottimo della funzione obiettivo del problema di Pricing è minore o uguale a 1.

3. Effettuare il round-up della soluzione frazionaria ottenuta.

3.1.1 Determinare i cutting patterns iniziali

Il primo passo, data un'istanza del problema, consiste nell'individuare un insieme di cutting pattern che andranno a comporre la matrice A_R, cioè rappresentano i cutting patterns ammessi nel primo problema ristretto individuato.

```
1 def determineInitialCuttingPatterns1(L, listOfModule):
2     m = len(listOfModule)
3     initialLenght = L
4     A_R = []
5     A_j = []
6     for j in range(0, m):
7         value = floor(L / listOfModule[j])
```

```
        for i in range(0, m):
            if i == j:
                A_j.append(value)
            else:
                A_j.append(0)
        A_R.append(A_j)
        A_j = []
    return A_R
```

Il metodo considera ciascun modulo $j = 1, ..., m$, per ciascuno calcola il valore $\left\lfloor \frac{L}{l_j} \right\rfloor$ e crea A_j come segue:

$$A^j = \begin{bmatrix} a_1 = 0 \\ \vdots \\ a_{j-1} = 0 \\ a_j = \left\lfloor \frac{L}{l_j} \right\rfloor \\ a_{j+1} = 0 \\ \vdots \\ a_m = 0 \end{bmatrix}$$

dunque la matrice A_R iniziale sarà una matrice diagonale come segue:

$$A_R = \begin{bmatrix} \left\lfloor \frac{L}{l_1} \right\rfloor & 0 & \dots & 0 \\ 0 & \left\lfloor \frac{L}{l_2} \right\rfloor & \dots & 0 \\ \vdots & \vdots & \ddots & \vdots \\ 0 & 0 & \dots & \left\lfloor \frac{L}{l_m} \right\rfloor \end{bmatrix}$$

evidentemente questo metodo utilizzato per inizializzare la matrice A_R non è l'unico metodo possibile, ne è il metodo migliore.

3.1.2 Iterazione

3.1.2.1 Risolvere il problema ristretto

```
def resolveSubproblem(listOfDemand, A_R):
    Y = []
    X = []

    #Creazione del modello
    primal = gb.Model("Primal")

    #Creazione delle variabili
    xVar = primal.addVars(len(A_R), vtype=GRB.CONTINUOUS, name ="x"
    , lb=0.0)
    primal.update()

    #for i in range(len(cutScheme)):
    #    print(xVar[i])
```

```
14
15      #Funzione obiettivo:
16      primal.setObjective(xVar.sum(), GRB.MINIMIZE)
17
18      #Vincoli del problema:
19
20      npMatA_R = np.matrix(A_R).transpose()
21     # print(npMatA_R)
22     # print(npMatA_R[0,0], npMatA_R[0,1], npMatA_R[0,2], npMatA_R
        [0,3])
23     #print(len(listOfDemand), len(cutScheme))
24      demands = primal.addConstrs(sum(xVar[j] * npMatA_R[i,j] for j
        in range(len(A_R))) >= listOfDemand[i] for i in range(len(
        listOfDemand)))
25
26
27
28      # Solver
29      print("Problema")
30      primal.update()
31      primal.optimize()
32
33      status = primal.getAttr("Status")
34      if status == 3: #status = 3 means Infeasible Model
35          print("Problema Infeasible")
36          return primal, Y, X, isOpt, status
37
38
39   #  for v in primal.getVars():
40    #     print(v.varName, v.x)
41
42      # Stampa valore funzione obiettivo
43      print('Valore ottimo della funzione obiettivo: ', primal.objVal
        )
44
45      for v in primal.getVars():
46          X.append(v.x)
47
48      for constr in primal.getConstrs():
49          Y.append(constr.Pi)
50
51      print(primal.display())
52
53      return primal, Y, X, status
54
```

Il metodo semplicemente modella tramite l'API di Gurobi per Python il problema ristretto, dunque: inizializza il modello; crea un numero di variabili continue pari al numero di colonne della matrice A_R; setta la funzione obiettivo; setta i vincoli, operazione per la quale ci si appoggia alla libreria numpy per gestire più facilmente la matrice A_R e infine lancia la risoluzione del problema.

3.1.2.2 Risolvere il problema di Pricing

```
1 def resolvePricing(L, listObjectiveFunction, listInequity):
```

```
    A_j = []

    pricing = gb.Model("Pricing")

    #Creazione delle variabili
    xVar = pricing.addVars(len(listObjectiveFunction), vtype=GRB.
    INTEGER, name = "x",lb=0.0)

    #Funzione obiettivo:
    pricing.setObjective(sum(xVar[j] * listObjectiveFunction[j] for
     j in range(len(listObjectiveFunction))), GRB.MAXIMIZE)

    #Vincoli del problema:
    demands = pricing.addConstr(sum(xVar[j] * listInequity[j] for j
     in range(len(listInequity))) <= L)


    pricing.update()
    pricing.optimize()



    #Stampa valore della funzione obiettivo del duale del problema
    ristretto
    print('Funzione obiettivo del duale del problema ristretto: ',
    pricing.objVal)

    #Verifico se il valore della funzione obiettivo e' maggiore di
    1, se cio' e' verificato, A_j diventera' il nuovo cutting
    pattern da considerare
    if pricing.objVal > 1:
        for v in pricing.getVars():
            A_j.append(v.x)

    return A_j, pricing.objVal

```

Aanche in questo caso il metodo semplicemente modella tramite l'API di Gurobi per Python il problema di Pricing. Si noti che nel caso in cui il valore ottimo della funzione obiettivo del problema di pricing fosse > 1 allora il metodo ritorna il cutting pattern da aggiungere a A_R, altrimenti ritorna un vettore vuoto.

3.1.2.3 Condizione di terminazione

Se la risoluzione del problema di Pricing restituisce un vettore vuoto allora si esce dal ciclo, altrimenti si aggiunge il vettore come colonna a A_R e si itera.

```
def updateA_R(A_R, A_j):
    A_R.append(A_j)
    return A_R
```

3.1.3 Round-Up della soluzione frazionaria

A questo punto la soluzione ottenuta risolvendo il problema ristretto è la soluzione del rilassamento lineare del CSP, dunque è sufficiente effettuare il round-up di

questa soluzione per ottenere una soluzione approssimata per il CSP originale.

```
def roundUpSolution(X):
    objRoundedUp = 0
    for x_j in X:
        objRoundedUp += math.ceil(x_j)
    return objRoundedUp

```

3.2 Limitazione dell'implementazione

L'implementazione soffre della limitazione legata all'uso della licenza gratuita di Gurobi; dunque, se il problema cresce troppo di dimensione, circostanza che si verifica quando il metodo itera molte volte, il programma si interrompe segnalando che si è raggiunto il limite della dimensione del problema risolvibile con la licenza gratuita.
Si è osservato un comportamento anomalo con alcune istanze del problema; in particolare il fatto che, seppur come vedremo, orientativamente sono necessarie dalle 20 alle 40 iterazioni per raggiungere la soluzione ottima (valore che dipende da alcuni parametri, come vedremo), in alcuni casi le iterazioni continuano a crescere fino a rendere il problema troppo grande per essere risolto con la licenza gratuita, dunque si è imposto un limite massimo di iterazioni del problema pari a 500 iterazioni.

4 Simulazioni e Risultati

I risultati sono stati prodotti effettuando simulazioni variando 5 parametri:

- lunghezza del paper roll
- numero di moduli
- domanda media per modulo
- lunghezza media dei moduli
- varianza della lunghezza dei moduli

Per ogni valore di questi parametri sono state effettuate simulazioni su 20 istanze generate randomicamente e calcolando tramite l'algoritmo di Welford:

- media del numero di iterazioni (colonne generate)
- deviazione standard del numero di iterazioni
- media del tempo di esecuzione
- deviazione standard del tempo di esecuzione
- media dell'errore assoluto (differenza tra il round-up della soluzione frazionaria e la soluzione frazionaria)

- media dell'errore relativo

Si noti che non potendo conoscere il valore ottimo intero, non è possibile calcolare l'errore assoluto e l'errore relativo compiuto considerando come ottima la soluzione ottenuta dal round-up della soluzione frazionaria; si possono solo calcolare degli upper bound considerando la differenza tra la soluzione round-up e la soluzione frazionaria al posto di quella intera.
Si noti inoltre che per il calcolo dei risultati non sono state considerate quelle istanze anomale, cioè quelle per cui l'algoritmo non converge velocemente alla soluzione ottima, ma continua ad aggiungere colonne fino a raggiungere la dimensione limite del problema imposta dalla licenza gratuita di Gurobi. Questa pratica rende i risultati trovati ottimistici rispetto alle prestazioni reali e validi solo sulle istanze "non anomale".

4.1 Lunghezza del paper roll

Per osservare come la lunghezza del paper roll impatta sulle prestazioni del sistema si sono effettuate simulazioni mantenenedo fisse le lunghezze dei moduli e le corrispettive domande; dunque gli esperimenti sono stati svolti variando la lunghezza del paper roll da un minimo di 100 ad un massimo di 10000 a passi di 100.

In particolare si sono effettuate simulazioni fissando: il numero di moduli pari a 20; le lunghezze dei moduli a valori estratti randomicamente con media 100 e supporto in 1, ..., 200 e le domande fissate a valori estratti randomicamente con media 500 e supporto in 301, ..., 700.

Nella figura 1 possiamo vedere le prestazioni in funzione della lunghezza dei paper roll.
Notiamo anzitutto che:

- tempo di esecuzione e il numero di iterazioni necessarie sono grandezze strettamente correlate tra di loro, infatti lo shape dei loro andamenti è pressoché lo stesso e se andassimo a calcolare la correlazione tra i rispettivi vettori otterremmo $R = 0.9997$.

- se la lunghezza dei paper roll non è molto maggiore della lunghezza media dei moduli, l'algoritmo impiega in media più iterazioni per raggiungere il valore ottimo; tale risultato è controintuitivo perché fissate le lunghezze dei moduli, più il paper roll è lungo, più aumentano i cutting patterns ammissibili, dunque ci si aspetterebbe che l'algoritmo arrivi alla soluzione ottima più lentamente, ma non è così.

Nella figura 2 possiamo vedere come varia l'errore al variare della lunghezza dei paper roll.
Notiamo che mediamente sia l'errore assoluto che l'errore relativo crescono al crescere della lunghezza dei paper roll e inoltre riguardo l'errore relativo possiamo notare che al crescere della lunghezza del paper roll diventa più instabile, cioè aumenta la sua varianza.

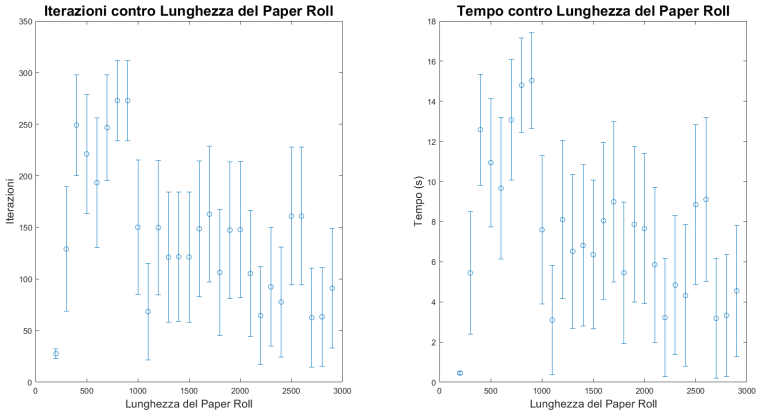

Figure 1: Prestazioni in funzione della Lunghezza dei Paper Roll

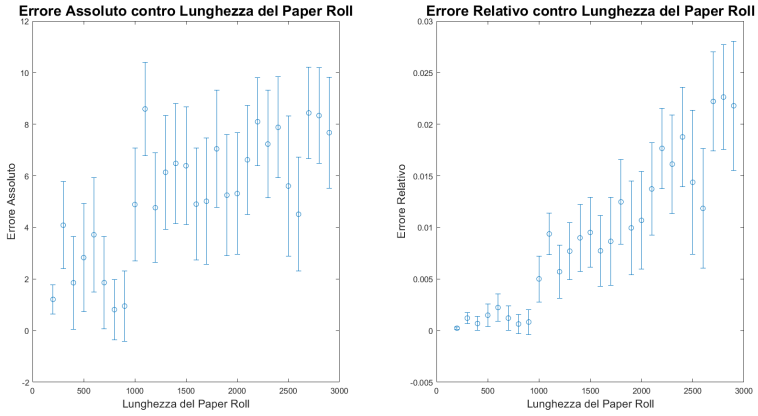

Figure 2: Errore in funzione della Lunghezza dei Paper Roll

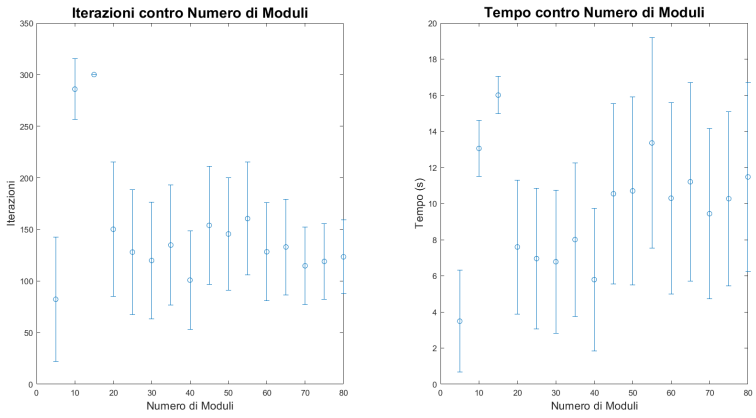

Figure 3: Prestazioni in funzione del Numero di Moduli

Si noti che nei risultati mostrati mancano quelli relativi al caso in cui la lunghezza del paper roll sia uguale a 100: ciò dipende dal fatto che in questo caso, esistono moduli di lunghezza maggiore rispetto a quella del paper roll, dunque non esistono cutting patterns ammissibili che generino moduli di tale lunghezza e di conseguenza la domanda relativa a questi moduli non può essere soddisfatta, rendendo il problema infeasible.

4.2 Numero di moduli

Per osservare come il numero di moduli impatta sulle prestazioni del sistema si sono effettuate simulazioni variando il numero di moduli da un minimo di 5 ad un massimo di 80, a passi di 5. In questo caso la lunghezza del paper roll è fissata a 1000.

Dalla figura 3 possiamo notare che il numero di iterazioni rimane pressoché invariato al crescere del numero di moduli, anche questo risultato è controintuitivo perché aumentando il numero di moduli si aumentano i possibili cutting patterns e dunque ci si aspetterebbe che fossero necessarie più iterazioni prima di trovare la soluzione ottima.

Per quanto riguarda l'errore, in figura 4 possiamo notare che sia la media che la varianza dell'errore assoluto crescono al crescere del numero di moduli; tale risultato era prevedibile, infatti nel paragrafo 2.2 abbiamo detto che l'errore assoluto è limitato superiormente dal numero di moduli m, dunque aumentando m stiamo "concedendo" all'errore assoluto di assumere un valore maggiore. Per quanto riguarda l'errore relativo si registra una crescita in media, ma quasi accennata.

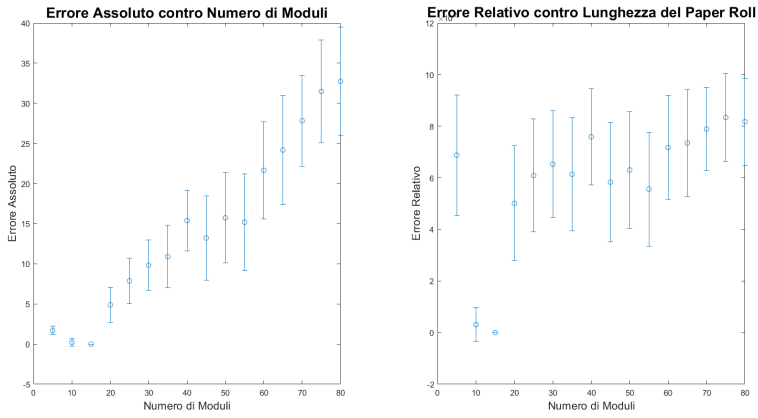

Figure 4: Errore in funzione del Numero di Moduli

4.3 Domanda Media

Per osservare come la domanda impatta sulle prestazioni del sistema si sono effettuate simulazioni variando la media e conseguentemente anche l'insieme di supporto della distribuzione da cui sono stati estratti i valori delle domande in modo da mantenere costante la sua varianza.

In particolare la domanda media è stata fatta variare da un minimo di 50, fino ad un massimo di 2000, a passi di 50.

Dalla figura 5 possiamo notare che quando la domanda media supera un certo valore, aumentarla ancora non impatta sull'esecuzione dell'algoritmo, dunque il numero di iterazioni e il tempo di esecuzione rimane pressoché invariato.

Per quanto riguarda l'errore, invece, dalla figura 6 possiamo notare che l'errore assoluto registra una leggerissima crescita al crescere della domanda media, mentre l'errore relativo si riduce seguendo un andamento iperbolico. Tale comportamento era prevedibile dalla teoria: infatti, un aumento della domanda media implica un aumento del valore delle variabili ottime, ma non un aumento del loro numero.

4.4 Media della Lunghezza dei Moduli

Per osservare come la media della lunghezza dei moduli impatta sul sistema si sono effettuate simulazioni variando la media e conseguentemente anche l'insieme di supporto della distribuzione da cui sono stati estratti i valori delle lunghezze dei moduli, ma non è stato cambiato l'ampiezza dell'intervallo da cui sono estratte, in questo modo la varianza è costante.

In particolare la lunghezza media è stata fatta variare da un minimo di 50, fino ad un massimo di 550, a passi di 25. L'ampiezza dell'intervallo è 100, dunque, ad esempio alla prima iterazione le lunghezze dei moduli sono estratte uniformemente dall'intervallo $[1, 101]$.

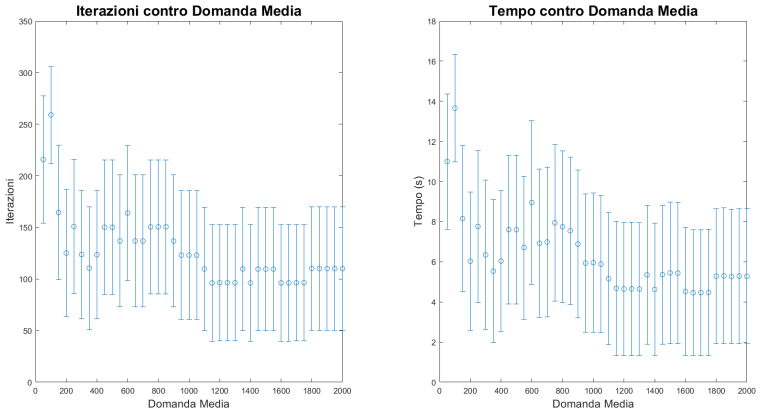

Figure 5: Prestazioni in funzione della Domanda Media

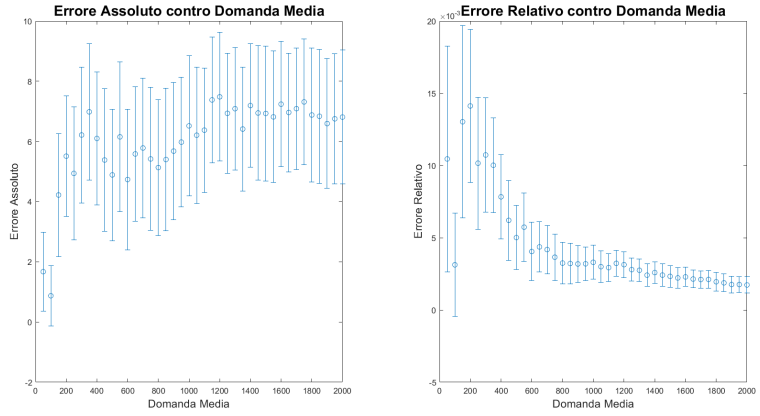

Figure 6: Errore in funzione della Domanda Media

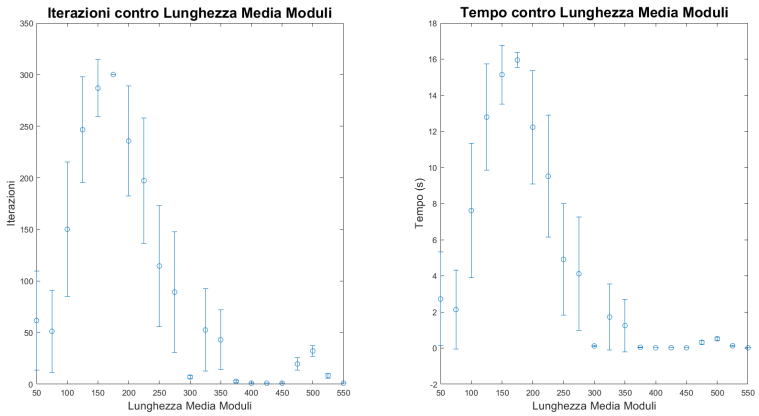

Figure 7: Prestazioni in funzione della Media della Lunghezza dei Moduli

Dalla figura 7 possiamo notare che al crescere della lunghezza media dei moduli diminuisce il numero di iterazioni necessari per giungere alla soluzione ottima, si noti in particolare che quando le lunghezze sono alte (425), si registra addirittura che già il primo problema di pricing testimonia l'inesistenza di un cutting pattern che deve essere inserito, cioè il primo problema ristretto contiene tutti i cutting patterns di cui si ha bisogno. Questo comportamento dipende dal fatto che considerata la lunghezza $L = 1000$ dei paper rolls, ogni paper roll è tagliato in modo da generare esattamente 2 moduli e questo è proprio il modo in cui è inizializzata la matrice A^R.

Per quanto riguarda l'errore, in figura 8 possiamo notare che l'errore relativo si riduce all'aumentare della lunghezza media dei moduli, ciò dipende dal fatto che se la lunghezza media dei moduli cresce, a parità di domanda aumenta il numero di paper rolls necessari e quindi il valore delle variabili ottime, ma non il loro numero.

4.5 Deviazione Standard della Lunghezza dei Moduli

Per osservare come la deviazione standard della lunghezza dei moduli impatta sul sistema si sono effettuate simulazioni varianzo l'insieme di supporto della distribuzione da cui sono stati estratti i valori delle lunghezze dei moduli, ma mantenendo costante il valore medio.

In particolare il valor medio è stato fissato a $m = 100$, dunque fissati alcuni valori della deviazione standard s, si sono ricavati gli estremi della distribuzione uniforme da cui sono state estratte, senza ripetizione, le lunghezze dei moduli. Gli estremi dell'intervallo della distribuzione uniforme $[a, b]$ sono stati calcolati

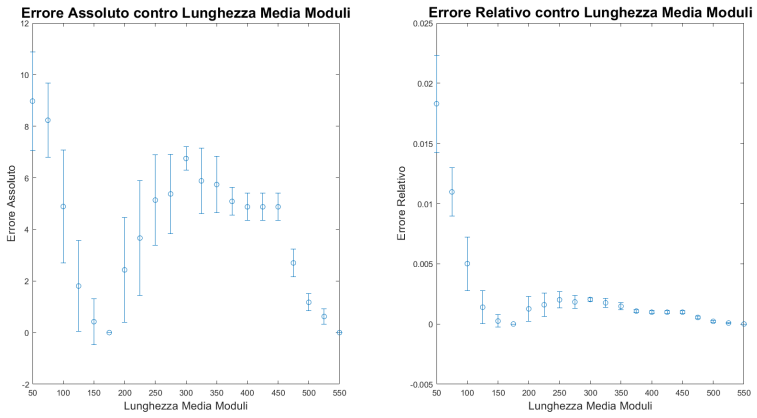

Figure 8: Errore in funzione della Media della Lunghezza dei Moduli

come segue:

$$a = \left\lfloor m - \frac{\sqrt{12s^2 + 1}}{2} + \frac{1}{2} \right\rfloor$$

$$b = \left\lceil m + \frac{\sqrt{12s^2 + 1}}{2} - \frac{1}{2} \right\rceil$$

la presenza dell'arrotondamento per difetto e per eccesso dipendono dal fatto che, in generale, i risultati di quelle espressioni sono valori frazionari, ma la routine utilizzata necessita che siano valori interi.

La figura 9 mostra che il numero di iterazioni necessarie per raggiungere l'ottimo diminuisce all'aumentare della deviazione standard della lunghezza dei moduli, fino a stabilizzarsi attorno ad un certo valore.

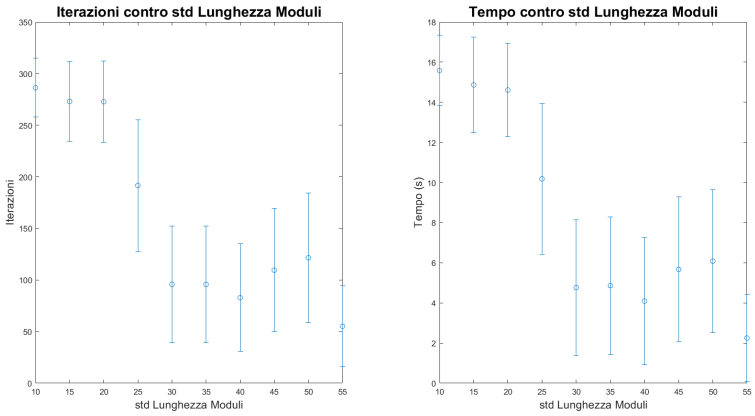

Figure 9: Prestazioni in funzione della Deviazione Standard della Lunghezza dei Moduli

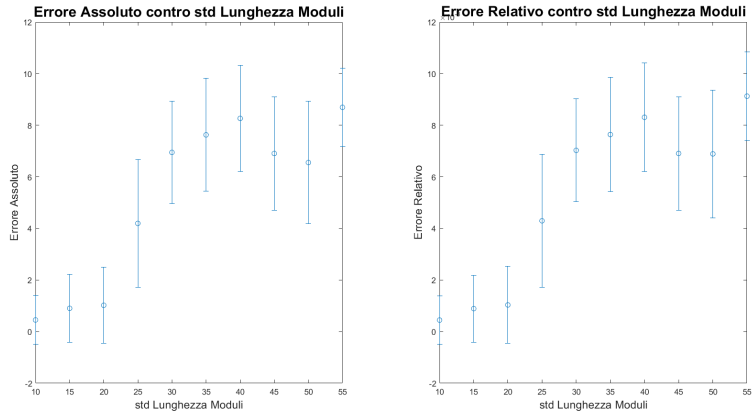

Figure 10: Errore in funzione della Media Deviazione Standard Lunghezza dei Moduli

5 Varianti

Come accennato nel paragrafo 3.1.1, il metodo proposto per inizializzare la matrice A_R non è l'unico metodo possibile, né il metodo migliore. A tal proposito sono state implementate e testate due varianti contenute nel file **cuttingStock.py** con nomi **determineInitialCuttingPatterns2** e **determineInitialCuttingPatterns3**.

5.1 determineInitialCuttingPatterns2

La prima variante riprende il metodo presentato nel paragrafo 3.1.1 in cui, abbiamo detto, si considera ciascun modulo $j = 1, ..., m$ e per ciascuno calcola il valore $\left\lfloor \frac{L}{l_j} \right\rfloor$ e crea A_j come segue:

$$A^j = \begin{bmatrix} a_1 = 0 \\ \vdots \\ a_{j-1} = 0 \\ a_j = \left\lfloor \frac{L}{l_j} \right\rfloor \\ a_{j+1} = 0 \\ \vdots \\ a_m = 0 \end{bmatrix}$$

La prima variante prevede che i moduli siano ordinati in ordine decrescente di lunghezza, quindi $l_1 > l_2 > ... > l_m$, quindi si considera ciascun modulo $j = 1, ..., m$: per il modulo j si calcola il valore $\left\lfloor \frac{L}{l_j} \right\rfloor$ e si pone $a_{jj} = \left\lfloor \frac{L}{l_j} \right\rfloor$. Il metodo del paragrafo 3.1.1 a questo punto passava al prossimo modulo; la variante, invece, continua a lavorare sullo stesso paper roll considerando lo scarto $scarto = L - \left\lfloor \frac{L}{l_j} \right\rfloor * l_j$ e considerando i moduli più corti del modulo j-esimo, quindi i moduli da j+1 a m. Il procedimento è iterativo e continua fino a che esiste un modulo che entri nello scarto prodotto dall'iterazione precedente.

ESEMPIO:
Supponiamo

- L = 20
- $l_1 = 10, l_2 = 7, l_3 = 4, l_4 = 1$

allora si ha: per j = 1: $a_{11} = \left\lfloor \frac{20}{10} \right\rfloor = 2$, lo scarto è 0 e dunque si passa al prossimo modulo, infatti $scarto = 20 - 2 * 10 = 0$

$$A^1 = \begin{bmatrix} a_{11} = 2 \\ a_{21} = 0 \\ a_{31} = 0 \\ a_{41} = 0 \end{bmatrix}$$

per j=2 si ha: $a_{22} = \left\lfloor \frac{20}{7} \right\rfloor = 2$, lo scarto è $scarto = 20 - 2 * 7 = 6$, dunque si riparte dallo scarto e dai moduli più piccoli di j=2, quindi da j+1=3 e si ha $a_{32} = \left\lfloor \frac{scarto}{l_3} \right\rfloor = \left\lfloor \frac{6}{4} \right\rfloor = 1$ e si produce un nuovo scarto: $scarto = 6 - 1 * 4 = 2$. Esiste un modulo che entra nel nuovo scarto prodotto? Si, il modulo l_4, dunque si continua e si ha $a_{42} = \left\lfloor \frac{4}{2} \right\rfloor = 2$, a questo punto lo scarto è 0, quindi sicuramente non posso aggiungere moduli, dunque abbiamo generato A^2 e possiamo passare a generare A^3.

$$A^2 = \begin{bmatrix} a_{12} = 0 \\ a_{22} = 2 \\ a_{32} = 1 \\ a_{42} = 2 \end{bmatrix}$$

Il codice è il seguente:

```
def determineInitialCuttingPatterns2(L, listOfModule):
    m = len(listOfModule)
    A_R = []
    A_j = np.ndarray.tolist(np.zeros(m))

    for j in range(0, m):
        A_j[j] = floor(L / listOfModule[j])

        remainder = L - A_j[j] * listOfModule[j]

        if j != m-1:
            for k in range(j+1, m):
                A_j[k] = floor(remainder / listOfModule[k])
                remainder = remainder - A_j[k]

        A_R.append(A_j)
        A_j = np.ndarray.tolist(np.zeros(m))

    return A_R
```

Il problema ristretto ottenuto generando A^R con questo metodo è sicuramente non peggiore di quello ottenuto generando A^R come descritto nel paragrafo 3.1.1, questo perché ogni cutting pattern ammissibile A^j generato con questo metodo produce al peggio gli stessi moduli prodotti dal cutting pattern ammissibile A^j generato col metodo descritto nel paragrafo 3.1.1.

5.2 determineInitialCuttingPatterns3

La seconda variante implementata segue le indicazioni proposte in [1] che fa riferimento a [2].

Il metodo consiste nel generare il cutting pattern search tree, cioè un albero in cui ogni cammino dalla radice ad una foglia identifica un cutting pattern. Se generassimo tutto l'albero, ovviamente, staremmo esplorando tutti i possibili cutting patterns e ciò sarebbe computazionalmente insostenibile, dunque sono state fatte delle ottimizzazioni per rendere questo metodo più veloce.

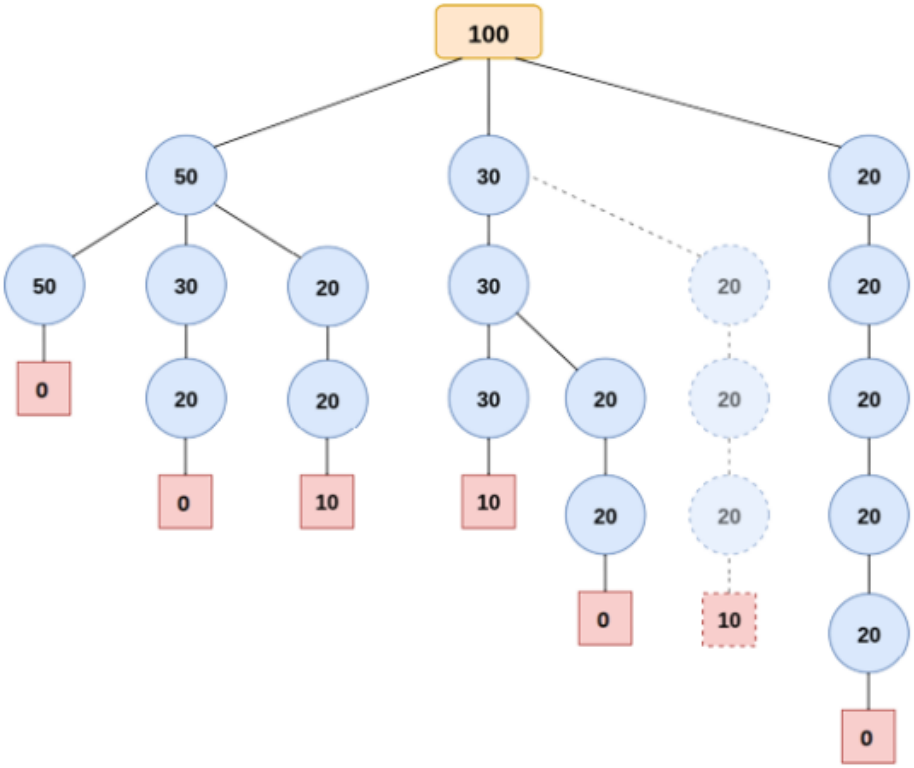

Figure 11: Cutting Pattern Search Tree

Nonostante queste ottimizzazioni il metodo risulta piuttosto lento, al punto che sarebbe stato impraticabile utilizzarlo nelle simulazioni.

Riproponendo l'esempio in [1]: supponiamo di avere paper rolls di lunghezza $L = 100$ e moduli di lunghezza $l_1 = 50, l_2 = 30, l_3 = 20$. Dunque generiamo il cutting pattern search tree visibile in figura 11.
La radice, in arancione, rappresenta la lunghezza del paper roll, i nodi interni rappresentano la lunghezza dei moduli e le foglie, in rosso, rappresentano lo scarto; ogni cammino dalla radice alle foglie rappresenta un cutting pattern.
Partendo dalla radice, la procedura include ricorsivamente un nuovo ramo per ogni modulo che può essere ottenuto dalla lunghezza residua del paper roll. Se non si possono ottenere più moduli e dunque non si possono creare nuovi rami, allora siamo arrivati ad una foglia e la lunghezza residua del paper roll rappresenta lo scarto di quel cutting pattern.

L'ottimizzazione adoperata per evitare di generare tutto l'albero e quindi per evitare che la procedura sia computazionalmente troppo onerosa, è quella di limitare il numero di cutting patterns generati. In particolare, detto M il numero massimo di cutting pattern generati, per ogni modulo i si calcola il

numero massimo di cutting patterns che iniziano con la generazione del modulo i:

$$\mu_i = \left\lfloor M \times \frac{m - i + 1}{X} \right\rfloor, \text{ con } X = \sum_{i=1}^{m} i$$

nell'esempio $M = 6$, quindi $\mu_1 = 3, \mu_2 = 2, \mu_3 = 1$; come conseguenza di ciò, si può vedere nella figura 11 che il terzo cutting pattern che inizia con il modulo di lunghezza 30 è tratteggiato, ad indicare che non viene generato.

Il metodo che implementa questa procedura genera uno alla volta i sottoalberi che hanno come radici i figli della radice dell'albero (cioè i nodi al livello 1), e poi li pone come figli della radice.

La generazione dei sottoalberi avviene attraverso una procedura ricorsiva:

```
1  def generateSubTree(tree, node, remainingLenght,
       listOfAvailableModules, nextIndex, cuttings, startModuleIndex,
       pruning, maxCuttingPatterns):
2      """
3      cuttings e' la lista del numero di moduli per ciascuna
       lunghezza ottenuta dal paper roll finora
4      startModuleIndex e' l'indice del primo modulo da considerare
       per tagliare il paper roll
5      nextIndex e' il prossimo indice del nodo dell'albero
6      pruning e' un booleano che abilita/disabilita il pruning
7      cuttingPatternsStartingWithModule e' una lista che in posizione
        i contiene il numero di cutting patterns che prevedono
8          come primo cut la generazione del modulo i-esimo
9      maxCuttingPatterns e'una lista che in posizione i contiene il
       numero massimo di cutting patterns che prevedono
10         come primo cut la generazione del modulo i-esimo
11     """
12     for i in range(0, len(listOfAvailableModules)):
13         if remainingLenght - listOfAvailableModules[i] >= 0:
14             globalModuleIndex = startModuleIndex + i
15             if(pruning and len(tree.leaves()) == maxCuttingPatterns
       ):
16                 canICutMore = (remainingLenght -
       listOfAvailableModules[len(listOfAvailableModules) - 1] >= 0)
17                 if not canICutMore:
18                     return nextIndex
19             currentCuttings = cuttings.copy()
20             currentCuttings[globalModuleIndex] = currentCuttings[
       globalModuleIndex] + 1
21             newNode = tree.create_node(currentCuttings, nextIndex,
       parent=node)
22             nextIndex = nextIndex + 1
23             nextIndex = generateSubTree(tree, newNode,
       remainingLenght - listOfAvailableModules[i],
       listOfAvailableModules[i:], nextIndex, currentCuttings,
       globalModuleIndex, pruning, maxCuttingPatterns)
24
25     return nextIndex
```

gli aspetti tecnici salienti della procedura sono:

- i nodi dell'albero sono numerati con indici crescenti assegnati secondo una visita in profondità dell'albero;
- per evitare di generare cutting pattern uguali (generati cambiando solo l'ordine con cui vengono estratti i moduli a partire dal paper roll), ogni nodo può estrarre moduli di lunghezza minore o uguale alla lunghezza del modulo estrarre in corrispondenza del padre, quindi non esisterà un cammino (con riferimento alla figura 11) del tipo *100->20->30->20->20->10*. Nell'implementazione ciò è realizzato passando da padre in figlio il parametro *listOfAvailableModules[i:]* che rappresenta la sottolista dei moduli disponibili al padre che il figlio può utilizzare.

References

[1] Computational Performance Evaluation of Column Generation and Generate-and-Solve Techniques for the One-Dimensional Cutting Stock Problem https://www.mdpi.com/1999-4893/15/11/394/pdf

[2] Suliman, Pattern generating procedure for the cutting stock problem. https://www.sciencedirect.com/science/article/abs/pii/S0925527301001347?via%3Dihub