Università degli Studi di Padova

Dipartimento di Ingegneria Dell'Informazione

Corso di laurea magistrale in Ingegneria Informatica

Analisi e implementazione di algoritmi e strategie per l'ottimizzazione del taglio con cesoia (nesting rettangolare)

Laureando: Fabio Grigolo

Relatore: Prof. Enrico Pagello

Correlatore: Walter Zanette

Anno Accademico 2012/2013

Indice

Elenco delle figure

Elenco delle tabelle

Introduzione

Sebbene l'avvento delle macchine per il taglio di lamiere mediante laser abbia catalizzato l'attenzione di tutto il mondo dello sviluppo hardware e software, le cesoie sono utensili ancora largamente utilizzate in ambito industriale e vantano attualmente più di cento costruttori sparsi per l'intero globo.

In campi quali automotive e pelletteria un fattore critico di successo per l'azienda è sicuramente rappresentato dal grado di ottimizzazione introdotto dagli algoritmi di taglio con cesoia. Tale ottimizzazione deve essere intesa sia come diminuzione del tempo necessario che come risparmio di risorse impiegate.

Se si pensa alle aziende del settore dell'automotive, dove all'interno di una singola giornata di lavoro si effettuano migliaia di tagli su lamiera, si può capire come un risparmio di tempo anche dell'ordine dei decimi di secondo per lamiera possa portare, nell'arco di un intero anno, ad un risparmio per l'azienda dell'ordine di milioni e milioni di euro.

L'ottimizzazione del processo si ha in due fasi: la fase di disposizione degli elementi sul foglio, detta anche fase di *nesting*, e la fase di taglio degli elementi, o *cutting*. Il problema del nesting è ampiamente affrontato in letteratura, ed il suo approfondimento sarà oggetto della sezione dedicata allo stato dell'arte. Per quanto riguarda il cutting, invece, non è stato possibile trovare esempi concreti in letteratura.

L'argomento centrale di questo lavoro è infatti lo sviluppo di algoritmi scalabili ed efficienti per ottimizzare la fase di cutting. Verrà fornita una definizione matematica del concetto di taglio ed in seguito verranno proposte alcune soluzioni al problema del cutting, delle quali verranno analizzate le prestazioni temporali e la qualità dei risultati.

Capitolo 1

Terminologia

In questa sezione si fornisce un elenco di definizioni dei termini utilizzati nelle pagine seguenti.

- **Semilavorato o foglio:** Rettangolo di lamiera che deve essere ulteriormente tagliato per ottenere i prodotti finiti.
- **Sfrido:** Parte di semilavorato che non diventerà un prodotto finito, è uno scarto di lavorazione.
- **Prodotto finito o lamierino:** Risultato ottenuti dalle incisioni dei semilavorati, è il prodotto che si desidera ottenere mediante il taglio.
- **Cesoia o cesoiatrice:** Macchina industriale mossa idraulicamente o meccanicamente, usata per lamiere di grande spessore o dimensione (massimo spessore della lamiera tagliabile è di circa 25-30 cm). Una descrizione più approfondita può essere trovata in [37].

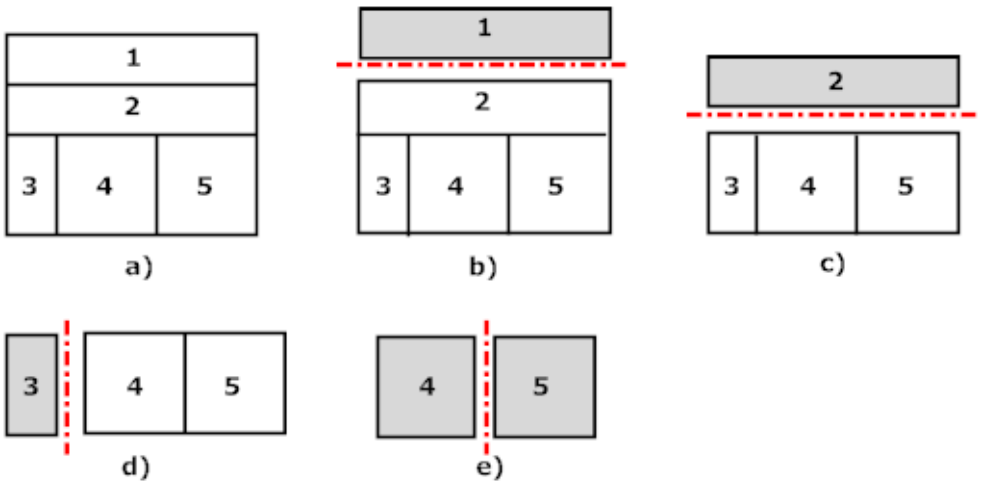


Figura 1.1: Procedura di taglio. **a)**. Semilavorato contenente 5 lamierini. Le figure successive **(b,..,e)** indicano una sequenza valida di tagli che consente di ottenere i cinque prodotti finiti desiderati.

Capitolo 2

Lavori precedenti

Il problema del nesting rettangolare è molto trattato in letteratura. Esso fa riferimento al classico problema di ottimizzazione lineare *CLP (Container Loading Problem)* bidimensionale. 2D-CLP infatti si propone di ottimizzare la disposizione di "pacchi" bidimensionali di forma rettangolare all'interno di un generico contenitore di forma rettangolare. La definizione matematica del problema risale agli inizi degli anni Sessanta del ventesimo secolo, grazie ai lavori di autori quali Gomory e Gilmore [15], [16].

2.1 Formulazione del Cutting Stock Problem di Gomory e Gilmore

Gomory e Gilmore formularono il problema come un problema di programmazione lineare intera, risolvibile mediante algoritmo del simplesso. I dati del problema sono: una lista di m oggetti rettangolari, ognuno dei quali ha una molteplicità q_j, $j = i, 2, ..., m$. Le x_i, $i = 1, ..., n$ rappresentano i blocchi di materia prima.

A questo punto il problema di programmazione lineare è il seguente:

$$min \sum_{i=1}^{n} c_i x_i$$

Soggetto ai vincoli

$$\sum_{i=1}^{n} a_{ij} x_i \geq q_j, \forall j = 1, ..., m$$

$$x_i \geq 0, \qquad x_i \; intero$$

In queste formule, i parametri a_{ij} indicano il numero di volte che l'oggetto j appare nel blocco i, mentre le c_i rappresentano l'utilizzo di materia prima.

La formulazione del problema è monodimensionale, ma può essere estesa anche al caso bidimensionale. Infatti la formulazione era utlizzata per minimizzare l'utilizzo della carta che si prelevava da rotoli di altezza costante.

La conversione al nesting su lamiera rettangolare è banale: infatti i "pacchi" diventano i lamierini e il contenitore diventa un foglio di lamiera. Una carrellata di tutte le varianti dei problemi CLP si può trovare in [9].

2.2 Categorizzazione dei problemi di Container Loading o Cutting Stock

L'articolo di Dyckhoff [9] cerca di categorizzare le varie categorie di problemi, e di individuarne proprietà e caratteristiche principali. Si giunse alla conclusione che i problemi di nesting possiedono quattro proprietà fondamentali, ognuna delle quali può avere diversi valori. Si può così procedere ad una classificazione dei problemi (anche se piuttosto qualitativa e non sempre facile da interpretare).

Author(s)	Year	Notion(s)	Discipline
Brown	1971	Packing, depletion	Computer Science
Salkin/de Kluyver	1975	Knapsack	Logistics
Golden	1976	Cutting stock	Industrial Engineering
Hinxman	1980	Trim loss, assortment	Operational Research
Garey/Johnson	1981	Bin packing	Combinatorial Optimization
Israni/Sanders	1982	Cutting stock, layout	Manufacturing
Rayward-Smith/Shing	1983	Bin packing	Mathematics
Coffman et al.	1984	Bin packing	Computer Science
Dowsland	1985	Packing	Operational Research
Dyckhoff et al.	1985	Trim loss	Management
Israni/Sanders	1985	Parts nesting	Production
Berkey/Wang	1987	Bin packing	Operational Research
Dudzinski/Walukiewicz	1987	Knapsack	Operational Research
Martello/Toth	1987	Knapsack	Mathematics
Rode/Rosenberg	1987	Trim loss	Engineering/Production
Dyckhoff et al.	1988	Cutting stock	Production

Tabella 2.1: La tabella mostra alcune applicazioni pratiche del problema del Container Loading Problem e delle sue varianti.

Le proprietà individuate da Dyckhoff e i valori che possono assumere sono indicati sotto.

1. Dimensionalità
   (a) 1 dimensione (1)
   (b) 2 dimensioni (2)
   (c) 3 dimensioni (3)
   (d) N dimensioni con N maggiore di 3 (n)

2. Tipo di assegnamento
   (a) Tutti gli *object* o contenitori ed una selezione degli *item* o contenuti (B)
   (b) Tutti gli item ed una selezione degli object (V)
3. Assortimento di object
   (a) Un solo object (O)
   (b) Più object tutti uguali (I)
   (c) Più object non tutti uguali (D)
4. Assortimento di item
   (a) Molti item di un ristretto gruppo di tipologie (F)
   (b) Molti item di un ristretto gruppo di tipologie (M)
   (c) Molti item di un ampio gruppo di tipologie (R)
   (d) Item della stessa tipologia (C)

Secondo questa categorizzazione si nota che possono esistere $4 * 2 * 3 * 4 = 96$ possibili tipologie del problema di CLP. Il problema del nesting su lamiera rientra nella categoria 2/V/D//.

Lo scopo della classificazione è quello di generalizzare algoritmi risolutivi: infatti se un algoritmo produce buoni risultati per un problema di tipo 3/B/D// molto probabilmente darà buoni risultati se applicato a qualsiasi problema della stessa categoria.

Notion	Belongs to type
(Classical) knapsack problem	1/B/O/
Pallet loading problem	2/B/O/C
More-dimensional knapsack problem	/B/O/
Dual bin packing problem	I/B/O/M
Vehicle loading problem	1/V/I/F, or 1/V/1/M
Container loading problem	3/V/l/, or 3/B/O/
(Classical) bin packing problem	1/V/I/M
Classical cutting stock problem	1/V/I/R
2-dimensional bin packing problem	2/V/D/M
Usual 2-dimensionalcutting stock problem	2/V/I/R
General cutting stock or trim loss problem	1///, 2///, or 3///
Assembly line balancing problem	1/V/I/M
Multiprocessor schedulingproblem	1/V/I/M
Memory allocation problem	I/V/I/M
Change making problem	1/B/O/R
Multi-period capital budgeting problem	n/B/O/

Tabella 2.2: Categorizzazione di alcuni problemi CLP classici utilizzando le proprietà individuate da Dyckhoff

.

Scheithauer [34], dimostrò che il problema del nesting è NP-Hard. Nel corso degli anni Novanta del ventesimo secolo sono state proposte diverse soluzioni basate su algoritmi metaeuristici. Gehring e Bortfeldt hanno proposto diversi approcci: un algoritmo genetico [12] a cui seguì una variante ibrida [13] e una parallela [14].

2.3 Algoritmo di Gehring, Bortfeldt, Mack

Gehring, Bortfeldt e Mack proposero nel 2003 una versione dell'algoritmo parallelizzata basata su Tabu Search [4].

Essi forniscono, nel loro articolo, la seguente definizione di nesting:

Definizione 2.1 (Nesting)**.** Dato un semilavorato contenitore e dei prodotti finiti, si ha un *nesting* se:

- Ogni prodotto finito è posizionato completamente all'interno del semilavorato;
- Non ci sono sovrapposizioni tra i prodotti finiti;
- Ogni prodotto finito è disposto parallelamente ai lati del semilavorato.

L'algoritmo di Gehring, Bortfeldt e Mack è composto di tre elementi:

1. Una euristica di base che ha lo scopo di creare un nesting completo;
2. Un algoritmo di tabu search sequenziale;
3. Una fase di parallelizzazione che si effettua lanciando diverse istanze di tabu search contemporaneamente.

2.3.1 Euristica di base

Utilizzando l'euristica il semilavorato viene riempito di prodotti finiti sfruttando diverse iterazioni. Ad ogni iterazione viene presa in considerazione solo una frazione del semilavorato, che viene chiamata *packing room*.
Una packing room è un'area vuota all'interno del semilavorato di dimensioni predefinite.
Nella prima iterazione tutto il semilavorato viene considerato come packing room.

Per riempire la packing room sono prese in considerazione solo conbinazioni di lamierini molto semplici. Esse vengono chiamate *local arrangments*. I local arrangments possono essere di due tipi: 1-arrangment e 2-arrangment. Gli 1-arrangment sono combinazioni di prodotti finiti tutti dello stesso tipo, mentre i 2-arrangment consentono di combinarne due tipi.

Lo pseudocodice seguente illustra per sommi capi l'algoritmo:

1. Inizializzazione:
   Insieme delle scatole residue $BRes$: tutte le scatole
   Lista delle packing rooms $PrList$: tutto il container
   Indice della packing room ipr: 0
   La lista di stoccaggio $StList$: $\emptyset$

2. Determina la packing room $pcurr$ di volume minimo all'interno di $PrList$ e cancella $pcurr$ da $PrList$.

3. Per $pcurr$ inizializza l'arrangment list $ArrList = \emptyset$. Genera e valuta local arrangment per $pcurr$. Inserisci gli arrangment in ordine discendente rispetto la valutazione nella lista $ArrList$.

4. Se $ArrList$ è vuota, vai al passo 8.

5. Aggiorna l'indice della packing room $ipr = ipr + 1$. Inserisci la coppia $(pcurr, ArrList(1))$ come l'ipr-esimo elemento nella lista $StList$.

6. Inserisci le packing room residue per $pcurr$ e il local arrangment $ArrList(1)$ nella lista $prList$

7. Aggiorna l'insieme delle scatole residue $BRes$

8. Se la lista delle packing room non è vuota, torna al passo 2.

9. Stop.

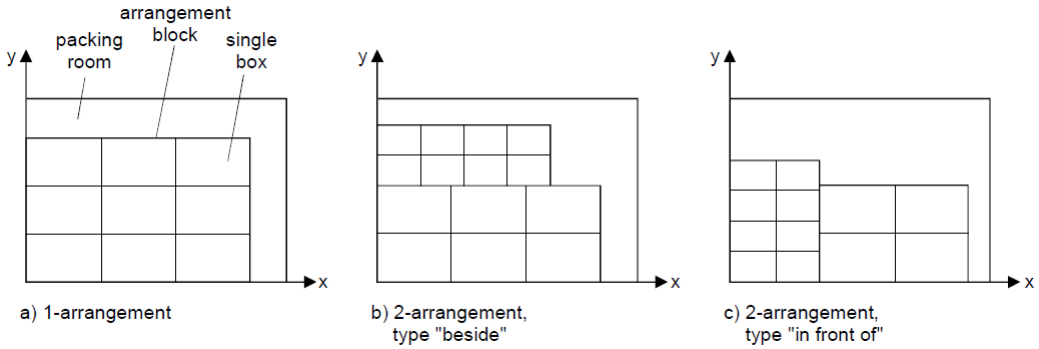


Figura 2.1: Possibili arrangment. Figura tratta da [4]

La figura 2.1 mostra i possibili arrangment che possono essere utilizzati per riempire la packing room. Una volta disposto un arrangment, la parte di semilavorato ancora vuota viene suddivisa in packing room più piccole e si ripete la procedura. L'algoritmo termina quando non ci sono più oggetti da disporre o quando finisce lo spazio.

2.3.2 Codifica delle soluzioni valide

Prima di passare alla fase di applicazione di Tabu Search, è necessario codificare le soluzioni valide. Per fare ciò si introduce il vettore Ps, cioè il *Packing sequence vector*. La posizione i-esima del vettore indica una packing room riempibile. Il vettore contiene un puntatore ad un arrangment e la dimensione dell'arrangment nella packing room stessa.

2.3.3 Applicazione di Tabu Search

Tabu Search è un algoritmo metaeuristico che cerca di migliorare i risultati ottenibili mediante local search, che ha il limite di fermarsi su minimi locali. Infatti *Tabu search* mantiene in memoria alcune soluzioni incontrate lungo il suo percorso di minimizzazione. Nel caso la ricerca giunga ad un minimo locale, si può far ripartire la ricerca stessa da uno dei valori salvati in precedenza. Un approfondimento dell'algoritmo è presente in [19].

2.3.4 Parallelizzazione dell'algoritmo

Esistono tre possibili modalità di parallelizzazione per algoritmi iterativi metaeuristici:

- Parallelizzazione delle operazioni all'interno di una singola iterazione;
- Decomposizione del dominio del problema o dello spazio di ricerca;
- Creazione di thread multipli di ricerca, con vari gradi di sincronizzazione e cooperzione.

In questo caso viene utilizzato il terzo tipo di approccio.
Una instanza del problema è affidata ad ogni thread. Ogni thread risolve il problema utilizzando TSA.
I parametri di TSA sono però diversi per ogni thread, in modo da esplorare la maggior varietà possibile di soluzioni.

Una volta applicato TSA, i thread operano uno scambio delle soluzioni. Le soluzioni ottenute da altri thread possono essere utilizzate come punto di partenza per ulteriori ricerche.

Prima di operare l'algoritmo parallelo spiegato precedentemente, è necessaria una fase sperimentale in cui si valuta quali siano i parametri migliori per TSA. I migliori set di parametri vengono quindi distribuiti ai vari thread.

Un altro parametro che incide molto sulle prestazioni dell'algoritmo è la frequenza di comunicazioni per lo scambio di soluzioni tra thread. Ricordiamo che gli algoritmi metaeuristici sono legati a due principi:

- *Exploitation*, cioè la ricerca di soluzioni nell'intorno di una soluzione già trovata;
- *Exploration*, la ricerca di nuove soluzioni in aree del dominio non ancora esplorate.

In un algoritmo iterativo parallelo è buona norma favorire l'exploration inizialmente per poi invece focalizzarsi sulla fase di exploitation nelle iterazioni finali. Chiaramente un maggior numero di comunicazioni favorisce l'exploitation a scapito dell'exploration, in quanto dopo una sincronizzazione i thread riprendono la

loro ricerca a partire dalla migliore soluzione ottenuta sino a quella iterazione da tutti i thread.

La configurazione più indicata prevede quindi una minore frequenza di comunicazioni nelle fasi iniziali e poi una frequenza sempre maggiore quando ci si avvia alle ultime iterazioni.

A partire dai risultati ottenuti dall'applicazione dell'algoritmo sviluppato in [4], si vuole estendere il problema passando alla fase successiva, quella del taglio. Nelle sezioni successive verrà introdotta una formulazione matematica del problema e verranno proposte alcune soluzioni ad esso.

Capitolo 3

Aspetti pratici del taglio con cesoia

Per capire il funzionamento degli algoritmi spiegati di seguito è necessario fornire una introduzione sul funzionamento della cesoia. Una cesoia è una macchina composta da una lunga lama che sovrasta un banco.
Una lamiera rettangolare deve essere suddivisa in sotto-lamierini, che sono i prodotti finiti desiderati. Solitamente parte della lamiera resta inutilizzata; tale parte si chiama sfrido.
Il semilavorato (lamiera rettangolare) viene posto sul banco. Successivamente la lama si abbassa sul banco fino ad incidere la lamiera rettangolare, ottenendo due nuovi oggetti rettangolari, che possono essere a loro volta semilavorati più piccoli o prodotti finiti.
Chiaramente per questioni economiche da un foglio di lamiera solitamente si cerca di ricavare il maggior numero possibile di prodotti finiti. Questo significa che non è poi così intuitivo per l'operatore capire qual'è la sequenza di taglio più indicata per rendere il lavoro più veloce e confortevole possibile.

Figura 3.1: Operazione di taglio di una lamiera con utilizzo di una cesoia idraulica. Fonte `http://www.hwupgrade.it/`

L'esperienza pratica di operatori più esperti ha fornito alcune linee guida per l'implementazione dell'algoritmo, in quanto essi hanno fornito alcuni parametri per valutare la sequenza di taglio. I parametri da minimizzare per ottenere una buona sequenza di taglio sono:

- **Numero di semilavorati non finiti presenti**. Questo perché un eccessivo numero di semilavorati non finiti causa confusione nell'operatore, che deve destreggiarsi tra una moltitudine di oggetti.
- **Rotazioni**. Ruotare un foglio di lamiera molto grande causa una perdita di tempo. Sarebbe utile ritardare le rotazioni il più possibile nel tempo.
- **Distanza tra tagli consecutivi**. Chiaramente tagliare parti di fogli lontane tra di loro causa una notevoli perdite di tempo, dovute alla movimentazione del semilavorato.

Figura 3.2: Alcuni costruttori di cesoie,fonte `http://www.directindustry.com/`

Capitolo 4

Algoritmi sviluppati

4.1 Introduzione

La trattazione di questo problema non trova riscontri in letteratura, quindi si è dovuto pensare ad un approccio completamente nuovo non avendo delle basi di partenza. Data la complessità del problema, si è pensato di utilizzare degli algoritmi approssimati, per poter ottenere una buona soluzione del problema in tempo polinomiale.
Il primo tentativo di fornire una soluzione è stato fatto cercando di traslare il problema del cutting in un problema su grafo, cercando in seguito di applicare gli algoritmi su grafi conosciuti in letteratura.
La seconda strada intrapresa (che si è alla fine rivelata vincente), è stata quella di utilizzare la metaeuristica GRASP. GRASP è una procedura iterativa generica di ottimizzazione combinatoria basata su soluzioni greedy. Ogni iterazione si articola nel modo seguente:

1. Generazione di una soluzione greedy leggermente randomizzata per il problema;
2. A partire da tale soluzione, costruire una sua neighbourhood[1];
3. Valutare il costo di ogni neighbour; se una neighbour ha costo minore della migliore soluzione trovata nelle iterazioni precedenti, tale soluzione diventa la nuova soluzione ottima.

4.2 Soluzioni basate su grafo

L'idea su cui si basa questo algoritmo è piuttosto semplice: mappare il problema su di un grafo, in cui i pesi apposti vari lati rappresentino il costo di un taglio, e applicare su questo grafo algoritmi di shortest path (ad esempio Dijkstra). Le difficoltà insite in questo approccio sono le seguenti:

[1] **neighbourhood**, che tradotto in italiano significa "vicinato". Termine molto comune quando si utilizzano algoritmi di ricerca locale. Con neighbourhood di una soluzione si intende un insieme di soluzioni di poco differenti dalla soluzione stessa.

- Costruzione del grafo stesso;
- Definizione di un modello di costo per i lati del grafo costruito.

4.2.1 Costruzione del grafo

Nella costruzione del grafo i lati rappresentano i tagli mentre i nodi rappresentano delle configurazioni di semilavorati e lamierini ottenuti mediante una sequenza di taglio. Naturalmente più sequenze di taglio possono portare ad una stessa configurazione.
Il grafo risultante sarà chiaramente un DAG (*Directed Acyclic Graph*), in cui ci sarà un nodo di partenza che possiede soli lati uscenti (corrispondente alla configurazione iniziale prima di effettuare qualsiasi taglio), ed un nodo di arrivo che possiede solo lati entranti (corrispondente alla soluzione finale). Naturalmente ogni percorso deve portare al nodo di arrivo.
Per costruire il grafo, la procedura etichetta ogni lato di ogni singolo lamierino con un identificatore univoco. Quando si effettua un taglio nel semilavorato, si tiene traccia degli identificativi dei lati che corrispondono al taglio stesso. La sequenza di lati tagliati viene utilizzata per etichettare il nodo. Prima di creare un nuovo nodo bisogna però verificare che la configurazione introdotta con il taglio non sia già stata incontrata lungo altri percorsi del grafo: se così fosse non viene creato un nuovo nodo, ma il lato che identifica il taglio viene collegato al nodo già esistente.

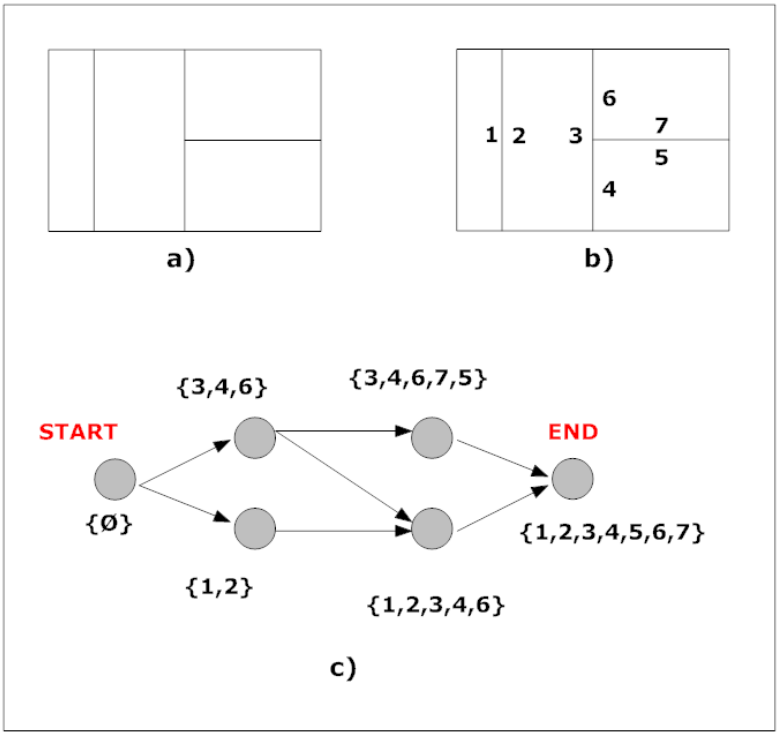


Figura 4.1: Esemplificazione grafica della costruzione del grafo. Figura a). Un semilavorato. b). I lati dei vari lamierini o prodotti finiti vengono etichettati in modo univoco. c) Grafo aciclico diretto risultante. Esso mette in evidenza che esistono più possibilità per giungere ad una stessa configurazione. Gli insiemi di numeri posti in corrispondenza di ogni nodo mostrano i lati che devono essere tagliati per giungere ad una particolare configurazione.

Come si può intuire da figura, la crescita del grafo è esponenziale rispetto al numero dei lamierini. Sperimentalmente si è verificato che con taglie del problema dell'ordine di 30-40 elementi il thread di esecuzione dell'algoritmo viene soppresso dal sistema operativo in quanto va ad occupare una quantità di risorse tali da compromettere il funzionamento del sistema operativo stesso.

4.3 GRASP: Generazione di una soluzione greedy

Tutte le procedure sviluppate successivamente hanno lo scopo di applicare la metaeuristica GRASP. Il primo passo per lo sviluppo dell'algoritmo consiste quindi nello sviluppo di una soluzione greedy. Per fare ciò dobbiamo per prima cosa definire cosa sono i tagli verticali e orizzontali.

Definizione 4.1 (Taglio verticale)**.** Sia R un semilavorato rettangolare, e sia h la sua altezza; sia L l'insieme dei lamierini contenuti in R. Allora un segmento S è un *taglio verticale* se e solo se sono valide le seguenti condizioni:

1. è orientato verticalmente (cioè le coordinate orizzontali dei suoi estremi coincidono);
2. la distanza tra i suoi due punti di estremo è pari ad h;
3. $\forall x \in L$, S non tocca alcun punto interno di x, al più si sovrappone ad uno dei segmenti che lo delimita;
4. $\exists x \in L$ per cui S è sovrapposto ad uno dei segmenti che lo delimita;
5. S non coincide con un bordo del semilavorato.

Analogamente possiamo dare la definizione di taglio orizzontale.

Definizione 4.2 (Taglio orizzontale)**.** Sia R un semilavorato rettangolare, e sia k la sua lunghezza; sia L l'insieme dei lamierini contenuti in R. Allora un segmento S è un *taglio orizzontale* se e solo se sono valide le seguenti condizioni:

1. è orientato orizzontalmente (cioè le coordinate verticali dei suoi estremi coincidono);
2. la distanza tra i suoi due punti di estremo è pari ad k;
3. $\forall x \in L$, S non tocca alcun punto interno di x, al più si sovrappone ad uno dei segmenti che lo delimita;
4. $\exists x \in L$ per cui S è sovrapposto ad uno dei segmenti che lo delimita;
5. S non coincide con un bordo del semilavorato.

Un taglio, se applicato ad un semilavorato, lo divide in due parti orizzontalmente o verticalmente. Ciascuno dei due oggetti risultanti può essere un semilavorato, un lamierino o una parte di sfrido. Ovviamente sui semilavorati devono essere applicati ulteriori tagli, mentre i lamierini e le parti di sfrido non vengono più prese in considerazione.

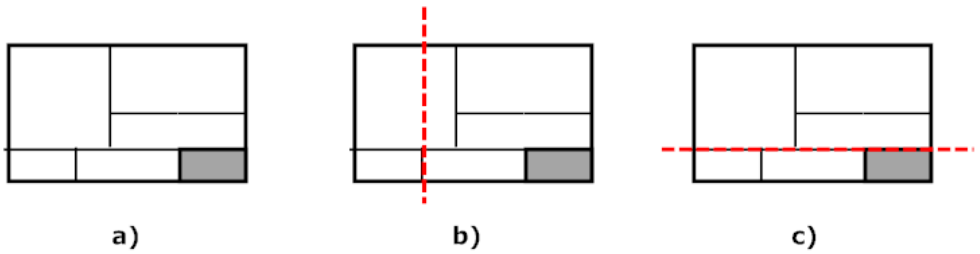


Figura 4.2: Nella figura **a)** è rappresentato un semilavorato con all'interno dei lamierini. La zona grigia in basso a a destra rappresenta lo sfrido. **b)**. La figura rappresenta un taglio verticale errato in quanto non soddisfa la condizione **3)** della definizione 4.1. **c)**. La figura rappresenta un taglio orizzontale in quanto tutte le condizioni della definizione 4.2 sono soddisfatte.

Un *cutting* è perciò una sequenza di tagli che riduce il semilavorato iniziale ad un insieme di lamierini e prodotti di scarto residui (sfrido).

Dato che l'input del problema è dato dalle coordinate del semilavorato e dei lamierini in esso contenuti, possiamo ricavare tutti i tagli nel semilavorato utilizzando le definizioni sopra citate.

Nella procedura seguente S indica il semilavorato, L l'insieme dei lamierini, H la lista dei lati orizzontali e V la lista dei lati verticali. Le coordinate di altezza e lunghezza saranno espresse come $g.X.min$, $g.Y.max$ eccetera, dove g è un generico oggetto rettangolare, X o Y indicano l'orientamento verticale e orizzontale, min e max indicano i valori massimo e minimo in altezza o lunghezza.

La procedura *SideDetection* illustrata di seguito ha lo scopo creare due insiemi H e V: il primo conterrà tutti i lati orizzontali di ogni lamierino, mentre V conterrà i lati verticali. L'utilità di raggruppare i lati secondo l'orientamento sarà più chiara in seguito.

SideDetection(S,L)
$H \leftarrow \emptyset$
$V \leftarrow \emptyset$
for each $l \in L$ **do**
for each side r in l **do**
if $(r.Y.min = r.Y.MAX)$
if $r.Y.min \neq S.Y.min$ **and** $r.Y.min \neq S.Y.max$
$H \leftarrow H \bigcup \{r\}$
else
if $r.X.min \neq S.X.min$ **and** $r.X.min \neq S.X.max$
$V \leftarrow V \bigcup \{r\}$
return H,V

A partire dagli insiemi ottenuti al passo precedente, effettuiamo due ordinamenti, uno per H e uno per V.
I lati di H vengono ordinati per coordinata y crescente, mentre gli elementi in V vengono ordinati secondo la coordinata x.
A questo punto si possono lanciare le routine di detection dei tagli.
Siano C_h e V_h gli insiemi che contengono i tagli:

```
HorizontalCutDetection(H,V)
    C_h ← ∅
    prev ← H[1].Y.min
    F ← H[1]
    for each s in H
        if s.Y.min = prev
            F ← F ⋃ s
        else
            if non ci sono tagli verticali k ∈ V
            tali che k.Y.min < prev e k.Y.max > prev
                C_h ← C_h ⋃ F
                F ← s
                prev ← s.Y.min
    return C_h
```

La routine raggruppa i lati orizzontali in sottoinsiemi di lati con la stessa coordinata y^*. In seguito controlla che non esistano lati verticali che abbiano un estremo con coordinata $y_1 < y^*$ e uno con coordinata $y_2 > y^*$. Infatti, se così fosse si violerebbe il punto 3 della definizione 4.2 e il taglio orizzontale trovato non sarebbe consistente.

Una routine analoga viene usata per la detection dei tagli verticali.
Chiaramente da un elemento C_h possiamo ricavare il taglio corrispondente in tempo $O(1)$ in quanto il taglio ha coordinata y uguale a quella di uno qualsiasi dei lati contenuti in un elemento di C_h. La lunghezza del taglio è invece pari a quella del semilavorato corrispondente.
A questo punto possiamo costruire la routine di scelta greedy della sequenza di tagli.

Sia W la lista dei tagli, *Slist* la lista dei semilavorati, la procedura è la seguente:

```
GreedySequence(Slist,H,V,W)
    K ← ∅
    J ← ∅
    for each S in Slist
        C_h ← HorizontalCutDetection(H, V)
        V_h ← VerticalCutDetection(H, V)
        K ← K ⋃ C_h
        J ← J ⋃ V_h
    scegli un taglio c in K o J
    W ← W ⋃ c
    rimuovi da SList il semilavorato tagliato nella riga precedente
        con il taglio c
    verifica se il taglio introduce nuovi
        semilavorati e aggiungili alla Slist
    aggiorna H e V rimuovendo i lati sovrapposti al taglio effettuato
    if Slist.size ≠ 0
        return GreedySequence(Slist,H,V,W)
    else
        return W
```

W, l'output della procedura, contiene una lista di coordinate di tagli verticali e orizzontali che portano a una soluzione del problema. La sezione successiva illustrerà i criteri di scelta del taglio da inserire in w partire dalle due liste K e J.

4.3.1 Criteri di scelta dei tagli

L'algoritmo GRASP prevede che la generazione della soluzione greedy non sia deterministica ma che le scelte siano leggermente randomizzate, al fine di esplorare più a fondo lo spazio delle soluzioni.
Ad ogni passo della costruzione deve essere creata la *Restricted Candidate List*, cioè una lista di possibili tagli da scegliere: tale lista contiene la scelta greedy più altri possibili tagli, tutti caratterizzati da un costo superiore a quello della soluzione greedy. Il costo deve essere comunque limitato entro una certa soglia.
Le modalità di costruzione della Restricted Candidate List influiscono pesantemente sull'efficacia dell'algoritmo, così come il criterio di scelta del taglio all'interno della RCL stessa.

Tiplogie di criteri di scelta utilizzati per costruire la RCL

Sono stati sviluppati in tutto tre criteri di scelta, uno basato su pesi e due basati su priorità. Il modello di costo basato su pesi tiene conto di tre grandezze fisiche:

- numero di semilavorati aggiuntivi introdotti dal taglio in questione;

- nel caso di rotazioni di fogli, grandezza del foglio ruotato.
- distanza dal taglio precedente.

Tutte e tre le grandezze vengono normalizzate (cioè ridotte ad un intervallo di valori compreso tra 0 ed 1), in modo da semplificare la procedura di weighting dei tre parametri. La funzione di costo diventa quindi del tipo:

$$A*NumeroSemilavorati+B*CostoRotazioni+C*DistanzaTaglioPrecedente$$

dove A,B,C sono tre valori interi su cui bisogna effettuare una operazione di tuning. Una volta assegnato un costo ad ogni taglio, basterà ordinare gli elementi per costo crescente e utilizzare i primi N (N è un parametro su cui dovrà essere fatto del tuning) per creare la *RCL*.

Anche se molto semplice ed intuitivo dal punto di vista teorico, questo modello è quello che ha dato i risultati peggiori dal punto di vista pratico: il problema è infatti trovare un set di parametri che dia buoni risultati con il maggior numero di configurazioni possibili. Tutte le configurazioni testate soffrivano di overfitting, cioè erano molto buone per alcune soluzioni e pessime per altre.

Il secondo approccio è basato su criteri di scelta prioritari. Si crea una gerarchia delle grandezze fisiche che influiscono sulla bontà della soluzione. Nel caso la grandezza fisica di primo livello risulti di egual valore per una coppia di tagli, si passa a quella di secondo livello e così via. I due criteri adottati funzionano alla stessa maniera, anche se valutano grandezze fisiche leggermente diverse. Il primo criterio valuta le seguenti grandezze, in ordine decrescente di priorità.

- Numero di semilavorati prodotti;
- Presenza di rotazioni;
- Distanza dal taglio precedente.

Per il secondo criterio le grandezze misurate sono queste (sempre ordinate secondo priorità decrescente):

- Numero di semilavorati prodotti;
- Presenza di rotazioni;
- Distanza del taglio dal bordo del semilavorato.

I due modelli sono del tutto equivalenti sotto il punto di vista del funzionamento. La scelta del modello migliore è stata fatta quindi sottoponendo i risultati

ottenuti a valutazioni di tipo pratico (cioè basate su metodi empirici) affidate ad esperti del taglio con cesoia. Il modello risultante migliore si è rivelato il primo.

I criteri illustrati consentono di definire una relazione di ordine per i tagli, che consentirà la costruzione della *Restricted Candidate List*. La sezione successiva illustrerà come avviene tale costruzione a seconda del criterio di scelta utilizzato.

4.3.2 Costruzione della Restricted Candidate List

Il metodo di costruzione della *Restricted Candidate List* dipende fondamentalmente dal criterio di scelta dei tagli utilizzato.

Caso con criterio basato su pesi

Sia N il numero massimo di tagli che possono essere inseriti nella RCL. Se i tagli disponibili sono in tutto K, chiaramente la RCL avrà taglia pari a *min(K,N)*. Come detto nella sezione precedente, in caso di criterio di scelta basato su pesi i tagli hanno un costo pari ad un numero reale maggiore di zero. Intuitivamente, la costruzione si basa sulla seguente procedura.

BuildRCL(maxRCLSize, cutList)
RCL $\leftarrow \emptyset$
i $\leftarrow 0$
while $i < min(maxRCLSize, cutList.Size)$
RCL $\leftarrow$ RCL $\bigcup$ cutList[i]
$i \leftarrow i + 1$
return RCL

Caso con criterio basato su priorità

La costruzione della restricted candidate list nel caso di criterio basato su priorità è leggermente più complessa. Dato che si desidera che le priorità utilizzate per ordinare i tagli vengano rispettate anche in questa fase, si è adottata la procedura illustrata sotto.

```
BuildRCL(maxRCLSize, cutList)
    C ← cutList[1]
    RCL ← C
    i ← 2
    while RCL.Size < maxRCLSize and i < cutList.Size
        K ← cutList[i]
        if C.Orientation = K.Orientation and
            C.SemilavAmount = K.SemilavAmount
            RCL ← RCL ⋃ K
        else
            return RCL
        i ← i + 1
    return RCL;
```

Nella procedura illustrata sopra, $maxRCLSize$ è la massima dimensione della RCL. Con questo criterio di costruzione, si vede che il primo elemento nell'ordine viene sempre scelto, e poi viene utilizzato come riferimento per l'inserimento per la valutazione dei tagli successivi.
Infatti vengono inseriti nella RCL solo i tagli che soddisfano le seguenti condizioni:

- Numero di semilavorati prodotti pari al numero di semilavorati prodotti dal taglio di costo minimo;
- Stesso orientamento del taglio di costo minimo;
- Se C_i è la posizione del taglio all'interno della lista, allora $C_i \leq maxRCLSize$

Quando la procedura incontra un taglio che non soddisfa i requisiti sopra indicati, termina e restituisce la RCL creata. Infatti data la natura prioritaria dell'ordinamento dei tagli, il fatto di trovare un taglio che non soddisfi i requisiti fa si che tutti i tagli successivi per costruzione non soddisfino i requisiti. Si può inoltre notare che la RCL non può mai essere vuota, in quanto il primo taglio è sempre inserito.

4.3.3 Scelta elemento nella Restricted Candidate List

Una volta costruita la *Restricted Candidate List*, ogni elemento che la compone può essere scelto con una certa probabilità. Supponiamo di dare agli elementi degli indici $1, 2, ...N$, dove l'elemento di indice 1 è quello con il costo minore. La probabilità di scegliere un elemento di indice $i \in [1, 2, ..., N-1]$ è

$$P = \frac{1}{k^{i-1}} - \frac{1}{k^i}$$

mentre la probabilità di scegliere l'elemento di indice N è

$$P = \frac{1}{k^{N-1}}$$

con $k \in \mathbb{N}, k \geq 2$. Il parametro k è un parametro dell'algoritmo GRASP su cui è necessario fare il tuning.

4.3.4 Valutazione del costo della soluzione greedy

Una volta costruita la soluzione greedy utilizzando le procedure analizzate nelle pagine precedenti, è necessario creare una funzione di costo in grado di calcolare la bontà della soluzione stessa. Anche in questo caso è necessario operare una distinzione tra modello basato su pesi e modello basato su priorità.

Modello basato su pesi

Anche in questo caso il modello basato su pesi consente una soluzione più semplice ed immediata rispetto al modello basato su priorità. Sia $C_1, C_2, ..., C_{N-1}, CN$ la lista di tagli che rappresenta la soluzione trovata, e $K_1, K_2, ..., K_{N-1}, K_N$ i costi associati ad ogni taglio, trovati utilizzando la somma pesata presentata in precedenza. Il costo totale K è intuitivamente rappresentato da

$$K = \sum_{i=1}^{n} K_i$$

Modello basato su priorità

L'algoritmo di costruzione della *Restricted Candidate List* richiede che tutti i tagli all'interno della lista abbiano lo stesso numero di semilavorati residui creati e lo stesso orientamento; questo implica che l'unico parametro su cui si può valutare un costo è il terzo, cioè la distanza dal taglio precedente (prima variante) o la distanza dal bordo del semilavorato (seconda variante).
La funzione di costo valuta la distanza totale percorsa per operare la sequenza di taglio. Sia $C_1, C_2, ..., C_{N-1}, CN$ la lista di tagli che rappresenta la soluzione trovata e $D_1, D_2, ..., D_{N-1}, DN$ i costi associati ad ogni taglio.
Nella prima variante il primo taglio ha costo sempre 0, quindi il costo totale della soluzione è dato da:

$$K = \sum_{i=2}^{n} D_i$$

Nella seconda variante la formula diventa

$$K = \sum_{i=i}^{n} D_i$$

4.4 Applicazione di GRASP

Una volta trovato un algoritmo efficiente per generare soluzioni greedy leggermente randomizzate, possiamo applicare l'euristica GRASP [31].
Il passo successivo alla costruzione della soluzione greedy è di individuare un algoritmo efficiente per la creazione di una neighbourhood.
In letteratura si possono trovare molti casi in cui la neighbourhood viene costruita operando degli swap in modo opportuno sulla soluzione greedy; alcuni esempi sono dati da [21], [8], [5].
Nel nostro caso si dovranno cercare delle nuove soluzioni create mediante swapping di elementi il cui costo sia minore di quello della soluzione di partenza.

4.4.1 Ricerca di swap validi

Nella definizione di nuove soluzioni introdotte mediante swap di elementi bisogna evitare di introdurre dei tagli non validi (cioè segmenti che non rispettano le condizioni dei teoremi 4.1 e 4.2).

Si dimostra banalmente con un esempio grafico che non si può effettuare lo swap tra tagli consecutivi che presentano orientamenti diversi (vedere figura seguente).

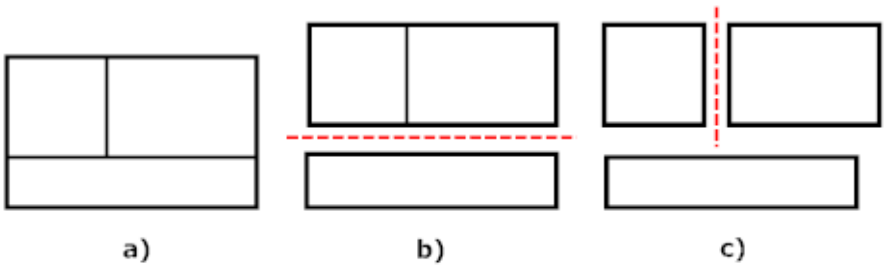


Figura 4.3: La figura mostra una sequenza di due tagli su di un semilavorato. E' evidente come l'inversione dell'ordine dei due tagli porterebbe il taglio verticale ad essere errato.

Inoltre si è evidenziato sperimentalmente che lo swap di tagli che producono un solo semilavorato (e hanno stesso orientamento) non porta a miglioramento

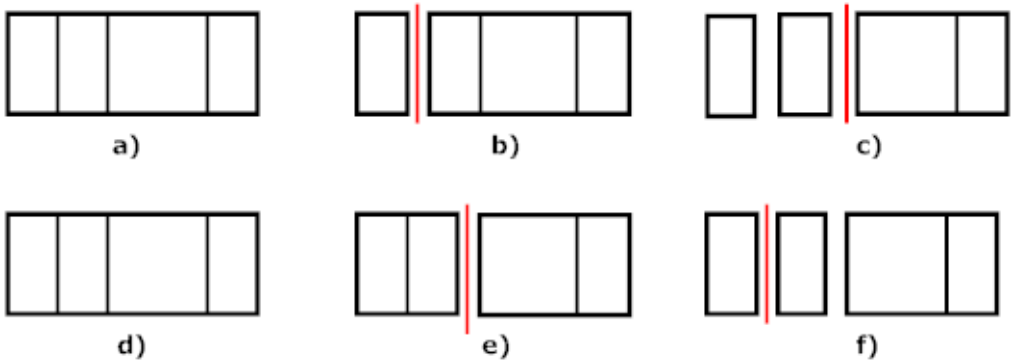


Figura 4.4: La figura mostra come lo swap dell'ordine di taglio violi il vincolo del minimo numero di semilavorati. Immagine **a**. Il semilavorato prima della sequenza di taglio. I due tagli successivi, **b** e **c**, danno luogo ad un solo semilavorato residuo. Questa sequenza non viola il vincolo della scelta del taglio che minimizzi i semilavorati residui. La sequenza successiva (**d-f**), viola il vincolo di priorità, in quanto il taglio scelto in **e** da' origine a due semilavorati.

della soluzione greedy, in quanto la nuova soluzione trovata viola la priorità di scelta (una spiegazione intuitiva viene fornita nell'immagine seguente).

Alla luce di quanto appare dalle illustrazioni seguenti, sono solo due i casi in cui è possibile effettuare uno swap.

- **Caso 1:** Due tagli consecutivi con lo stesso orientamento, ognuno dei quali crea zero semilavorati;

- **Caso 2:** Due tagli consecutivi con lo stesso orientamento, ognuno dei quali crea due semilavorati.

4.4.2 Costruzione della neighbourhood

Una volta stabilito quali sono gli swap ammissibili per la nostra soluzione, possiamo stabilire un criterio di costruzione della neighbourhood. La procedura seguente controlla quali coppie di tagli consecutivi possono essere scambiati i tra di loro. La posizione degli indici delle coppie di tagli scambiabili viene salvata nella lista C. In input viene fornita la sequenza dei tagli G. Per comodità si supponga che gli elementi di G siano forniti di due metodi:

- S, che restituisce il numero di semilavorati ottenuti effettuando il taglio;

- O, che fornisce l'orientamento del taglio (verticale od orizzontale).

```
SwapDetection(G)
    C ← ∅
    prev ← 1
    next ← 2
    while prev = G.length − 1
        SPrev ← G(prev).S
        SNext ← G(next).S
        if SPrev = 0 or SPrev = 2
            if SPrev = SNext and G(prev).O = G(next).O
                C ← C ⋃ < prev, next >
        prev ← prev + 1
        next ← next + 1
    return C
```

Ogni elemento presente in C è una coppia di elementi che possono essere scambiati di posto fra di loro mantenendo la nuova soluzione G' valida. Le nuove soluzioni che andranno a creare la neighbourhood sono tutte le soluzioni G' ottenute scambiando tra di loro due elementi secondo gli indici contenuti da C. A questo punto si possiedono tutti gli elementi per implementare l'euristica GRASP. Lo pseudocodice seguente mostra il funzionamento dell'euristica in dettaglio. In input vengono forniti:

- S, il semilavorato di partenza;
- α, intero che indica la lunghezza massima della *Restricted Candidate List*;
- R, il numero di iterazioni di GRASP;
- K, parametro che influenza la scelta degli elementi.

Per semplicità la costruzione della soluzione greedy viene eseguita con la procedura **GreedySolution** che richiede in input il semilavorato di partenza e la dimensione della RCL. La procedura **MinCostSolution** invece ritorna la sequenza di tagli di costo minore tra quella fornite in input.

```
GRASP(S,K,R,alpha)
    Best ← emptyset
    cost ← +∞
    for i ← 1, i ≤ R, i ← i + 1
        L ← GreedySolution(S,K,alpha)
        calcola gli swap ammissibili su L
        utilizza gli swap per calcolare una neighbourhood N
        N' ← N ⋃ L
        j ← MinCostSolution(N')
        if j.cost < cost
            Best ← j
            cost ← Best.cost
    return Best
```

4.5 Complessità computazionale dell'algoritmo

In questa sezione analizzeremo la complessità computazionale di ogni singola procedura al fine di valutare la complessità asintotica dell'algoritmo.

4.5.1 Complessità algoritmo di costruzione soluzione greedy

Consideriamo un semilavorato formato da m lati orizzontali e n verticali. La procedura di detection dei lati ha banalmente complessità computazionale $O(m+n)$. L'ordinamento delle liste dei lati, effettuato con quicksort, ha complessità ammortizzata $O(m \log m)$ per i lati orizzontali e $O(n \log n)$ per i lati verticali. La parte più onerosa dell'algoritmo è data dalla detection dei tagli. Nel caso peggiore la detection dei lati verticali ha costo $O(mn)$, e così anche la detection dei lati orizzontali. Nel *worst case*, che si ha quando ogni lato di un lamierino rappresenta un taglio distinto, si avranno $O(m+n)$ detection di tagli. Quindi la complessità computazionale data dalla costruzione della soluzione greedy è

$$O(m+n) + O(n \log n) + O(m \log n) + O(mn)O(m+n)$$

Considerando però che il numero di lati verticali e di lati orizzontali tende a coincidere, poniamo $n = m$, la formula precedente diventa:

$$O(n) + O(n \log n) + O(n^2)O(n) = O(n^3)$$

La complessità computazionale data dalla costruzione di una soluzione greedy è quindi $O(n^3)$

4.5.2 Costruzione della neighbourhood e valutazione costo

L'algoritmo di rilevazione delle coppie scambiabili consiste un ciclo *for* su di una sequenza di tagli. Dato che il numero di tagli è sicuramente limitato superiormente dal numero di lati, possiamo dire che la complessità della swap detection è $O(n)$. Ogni neighbour è costruita in tempo $O(n)$ e valutata il tempo $O(n)$. Il costo complessivo di costruzione della neighbourhood e valutazione del costo di ognuna è dunque dato da

$$O(n)(O(n) + O(n)) = O(n^2)$$

4.5.3 Conclusioni

Siano K le iterazioni di GRASP, allora l'algoritmo esegue K volte la procedura, essa ha costo $O(n^3)$ per la costruzione della soluzione greedy, a cui si aggiunge il costo dato dalla creazione e valutazione della neighbourhood, pari a $O(n^2)$. Quindi il costo dell'algoritmo è dato da

$$K(O(n^3) + O(n^2)) = KO(n^3) = O(Kn^3)$$

4.6 Parallelizzazione di GRASP

La struttura dell'algoritmo GRASP rende il codice facilmente parallelizzabile. Gli autori dell'algoritmo hanno analizzato in [1] due approcci diversi di parallelizzazione. Altri esempi di applicazione di GRASP parallelo sono trattati in [2], [28], [26] e [29]. In [33] viene trattata l'applicazione di GRASP combinata all'ottimizzazione GPU.

4.6.1 Parallelizzazione Semplice

La parallelizzazione semplice sfrutta il fatto che l'iterazione i-esima di GRASP non necessita dei dati di output ottenuti attraverso le altre iterazioni (precedenti e successive).
Ogni thread esegue una copia identica di GRASP; in questo modo le K iterazioni vengono suddivise equamente tra i vari thread al fine di diminuire il tempo totale di computazione. Quando tutti i thread hanno finito la fase di calcolo individuale, viene scelta la migliore soluzione tra tutte quelle prodotte da ogni thread. Se l'algoritmo viene suddiviso fra W thread, la complessitá asintotica diventa:

$$O\left(\frac{Kn^3}{W}\right)$$

4.6.2 Algoritmo OTOS (One Thread One Setting)

Il secondo algoritmo parallelo, a differenza del primo, propone una configurazione diversa di GRASP per ogni thread. I parametri che subiscono variazioni sono la lunghezza della restricted candidate list ed il parametro k[2]. Si é notato che nel 90% dei casi il costo della soluzione ricavata con questo algoritmo é minore o al piú uguale del costo ottenuto utilizzando la Parallelizzazione Semplice o il GRASP sequenziale.
Malgrado ciò, tale algoritmo ha mostrato dei tempi di calcolo proibitivi. L'utilità maggiore dell'utilizzo di quest'algoritmo si è avuta in fase di tuning dei parametri.

[2]Con parametro k in questo caso si fa riferimento a 4.3.3

Capitolo 5

Dettagli implementativi

L'applicazione è stata scritta nel linguaggio di programmazione *C#* su sistema operativo Windows 7. I vantaggi derivati dall'utilizzo di questo linguaggio sono evidenti soprattutto in fase di parallelizzazione, dove si è sfrutta la potenza di *.NET Framework 4.5*. Una trattazione approfondita di C# e *.NET Framework 4.5* è presente in [35], [27], [3]. Le librerie parallele di C# sono invece trattate in [11] e [23].

L'implementazione dell'algoritmo Grasp con thread replicati si è ottenuta collocando la procedura di GRASP all'interno del costrutto **Parallel.For**. Questo comando pone le istruzioni all'interno di un ciclo come il normale *for*. Le iterazioni non vengono però eseguite in maniera sequenziale, ma ognuna viene assegnata ad un thread. Perché tale parallelizzazione sia effettivamente vantaggiosa è necessario però che la routine all'interno del ciclo abbia un tempo di esecuzione relativamente lungo. Quando così non è i vantaggi dell'esecuzione parallela si perdono in quanto il tempo perso per cambi di contesto necessari al lancio dei thread rende l'esecuzione più lenta di quella sequenziale.

Per l'implementazione dell'algoritmo *One-Thread One-Setting* si è dovuto procedere in maniera differente in quanto ogni thread deve eseguire una istanza di GRASP con parametri differenti. In questo caso si è proceduto creando dei task, assegnando ad ognuno una procedura GRASP con parametri differenti. L'esecuzione dei task viene svolta in parallelo.

Capitolo 6

Analisi delle prestazioni

Le prestazioni sono state misurate su di un *Acer Travelmate P253-M* che dispone di processore *Intel Core i5-3210M* con clock di 2.5 GHz e 4GB di memoria RAM. Il sistema operativo presente sulla macchina è Windows 7 Professional. Attraverso le misurazioni si vuole analizzare se:

- Per uno stesso problema, il tempo necessario per effettuare ogni iterazione di GRASP tende ad essere costante;
- Se la parallelizzazione di GRASP implica uno speedup nell'esecuzione dell'algoritmo.

Le tabelle ed i grafici seguenti mostrano i dati rilevati su grasp sequenziale. I valori temporali sono espressi in millisecondi.

Taglia	Iterazioni di GRASP				
-	32	64	128	256	512
10	7	13	22	41	81
20	17	33	67	131	262
40	220	430	860	1667	3382
80	1386	2768	5503	10807	21859

Tabella 6.1: Tempi di computazione di GRASP (caso sequenziale), al variare del numero di iterazioni e della taglia dell'input

L'analisi dei dati della tabella mostra innanzitutto che il tempo di esecuzione non aumenta in modo lineare con l'aumentare della taglia. Questo dato conferma l'analisi teorica della complessità computazionale, dalla quale era emerso che il costo computazionale dell'algoritmo di costruzione della soluzione greedy è $O(n^3)$. I grafici seguenti mostrano il tempo necessario alla computazione di GRASP mantenendo la taglia dell'input costante e raddoppiando costantemente il numero di iterazioni.

I grafici mostrano chiaramente l'andamento lineare del tempo di computazione. Il lettore non si lasci ingannare dalla curva dall'andamento parabolico, in

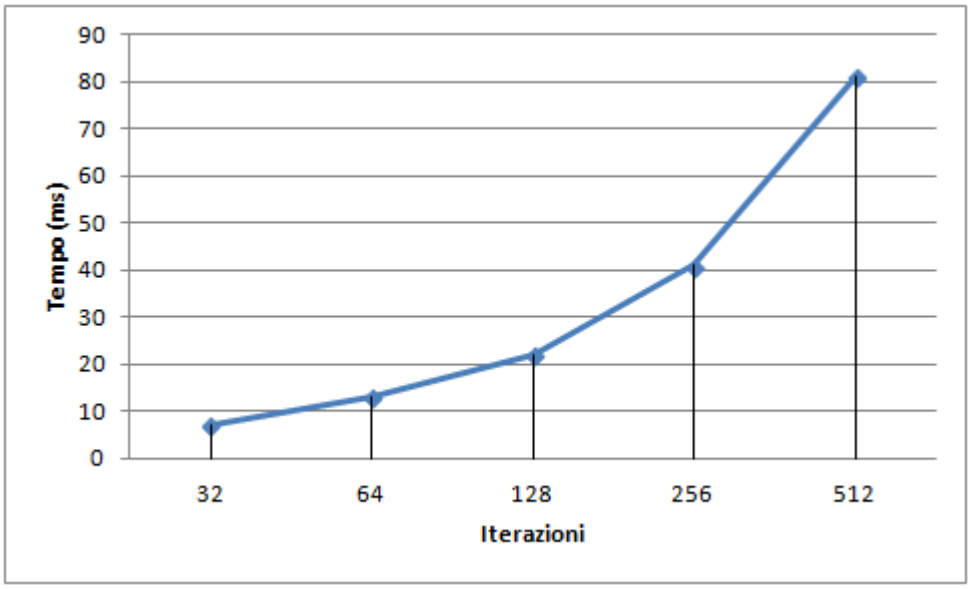


Figura 6.1: Tempi di computazione di GRASP su input di 10 elementi

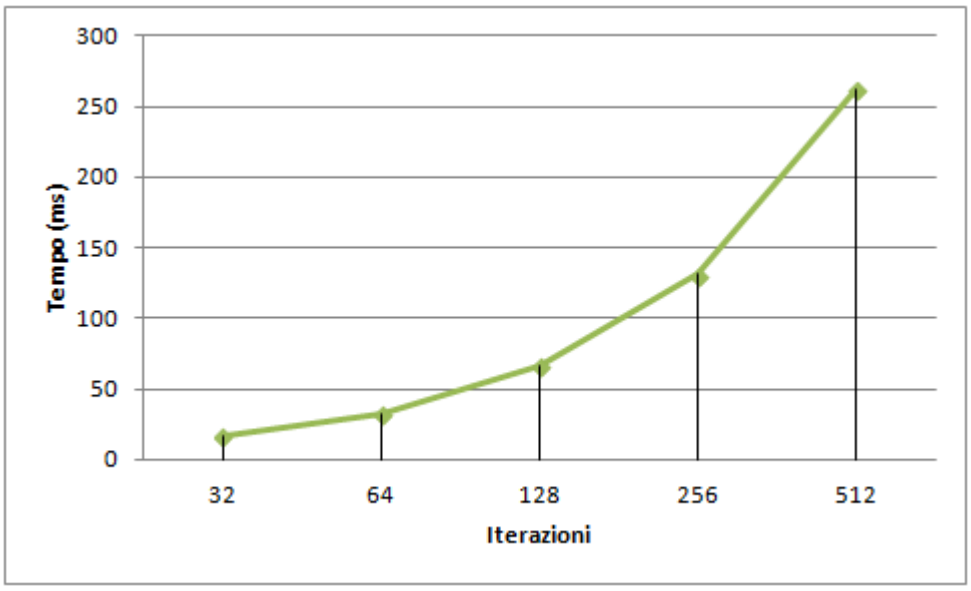


Figura 6.2: Tempi di computazione di GRASP su input di 20 elementi

quanto il grafico non presenta una scala lineare lungo l'asse orizzontale.
Questi grafici mostrano chiaramente che si può predire con buona precisione il tempo necessario all'elaborazione GRASP una volta determinato il tempo di computazione della soluzione greedy.

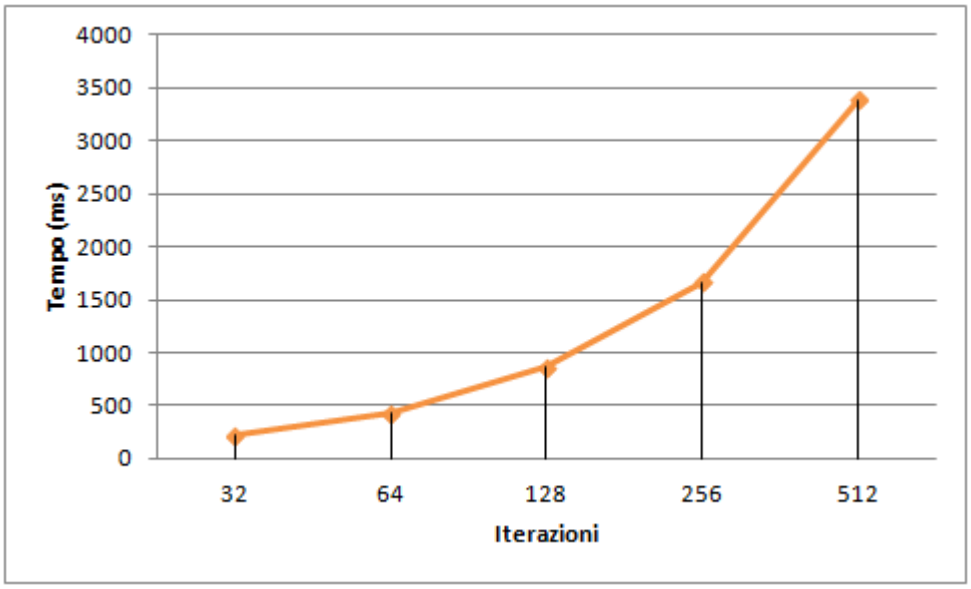


Figura 6.3: Tempi di computazione di GRASP su input di 40 elementi

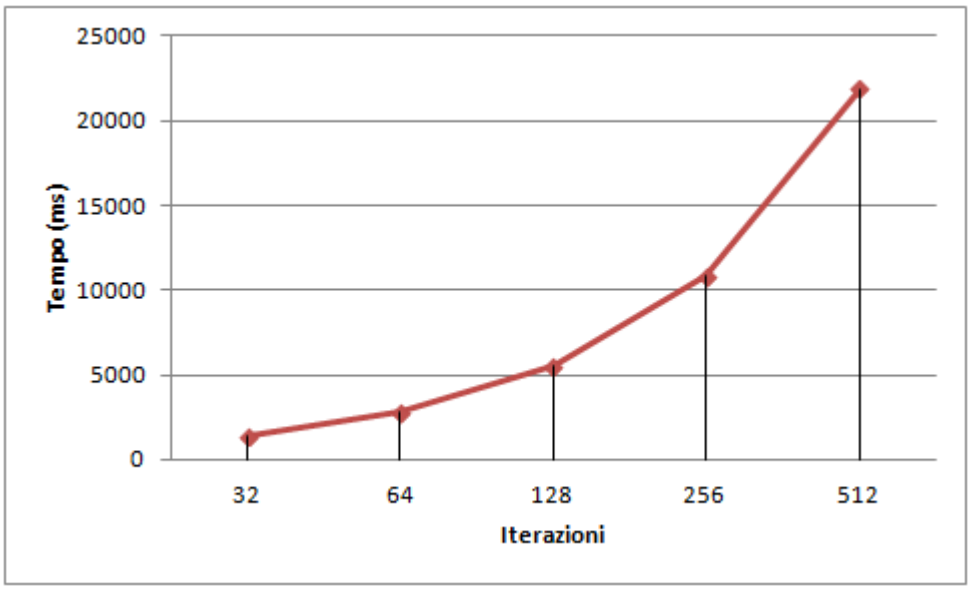


Figura 6.4: Tempi di computazione di GRASP su input di 80 elementi

6.1 Analisi delle prestazioni dell'algoritmo parallelo

In questa sezione verranno analizzate le prestazioni della versione parallelizzata dell'algoritmo e si forniranno alcuni commenti sui risultati ottenuti. Ricordiamo che la struttura dell'algoritmo parallelo è composta da diverse repliche dell'algoritmo sequenziale, ognuna delle quali affidate ad un thread. Il fatto che i vari thread operano in modo totalmente asincrono dovrebbe implicare un buon grado di parallelismo. Le tabelle seguenti mostrano i tempi rilevati utilizzando rispettivamente 2 thread e 4 thread.

Guardando le tabelle è facile notare che i tempi di esecuzione sono sicura-

Taglia	Iterazioni di GRASP				
-	32	64	128	256	512
10	4	7	14	28	60
20	11	22	41	88	168
40	135	275	513	1060	2157
80	879	1711	3342	6853	13371

Tabella 6.2: Tempi esecuzione dell'algoritmo GRASP parallelizzato su 2 thread

Taglia	Iterazioni di GRASP				
-	32	64	128	256	512
10	3	8	15	33	56
20	11	22	50	89	161
40	137	253	479	924	1824
80	817	1571	3112	6044	12072

Tabella 6.3: Tempi esecuzione dell'algoritmo GRASP parallelizzato su 4 thread

mente inferiori che nell'algoritmo sequenziale, anche se non è così chiaro quale sia il fattore di speedup ottenuto. Dato T_S (tempo di esecuzione dell'algoritmo sequenziale) e T_P (tempo di esecuzione algoritmo parallelo), ricordiamo che il fattore di speedup S di un algoritmo parallelo è un numero positivo che si ottiene dalla formula:

$$S = \frac{T_P}{T_S}$$

I grafici seguenti mostrano l'effettivo speedup ottenuto in ciascuno dei due casi:

Le tabelle, unitamente ai grafici, mostrano che i tempi di esecuzione con 2 o con 4 thread sono molto simili. Nel primo caso i tempi si avvicinano a quelli determinati in teoria, con un fattore di speedup che si attesta circa ad 1,5 su un massimo possibile di 2. Il caso a 4 thread invece non rispetta le aspettative teoriche, attestandosi ad un fattore di speedup di circa 1,7 su un massimo possibile di 4.

Per dare una spiegazione a questo fenomeno dobbiamo introdurre il concetto di *Overhead*. Sia T_k il tempo totale di esecuzione dell'algoritmo, mentre T_{ext} è la frazione di tempo in T_k dedicata ad operazioni quali cambi di contesto, I/O, buffering ecc. Possiamo definire l'*Overhead* O come il rapporto:

$$O = \frac{T_{ext}}{T_k} \cdot 100$$

Intuitivamente quindi l'Overhead indica la percentuale di tempo totale in cui la macchina non "lavora" all'algoritmo. Aumentando il numero di thread, a parità di iterazioni si ha che il tempo passato da ogni thread ad eseguire GRASP diminuisce mentre il tempo necessario al sistema operativo per creare i thread ed

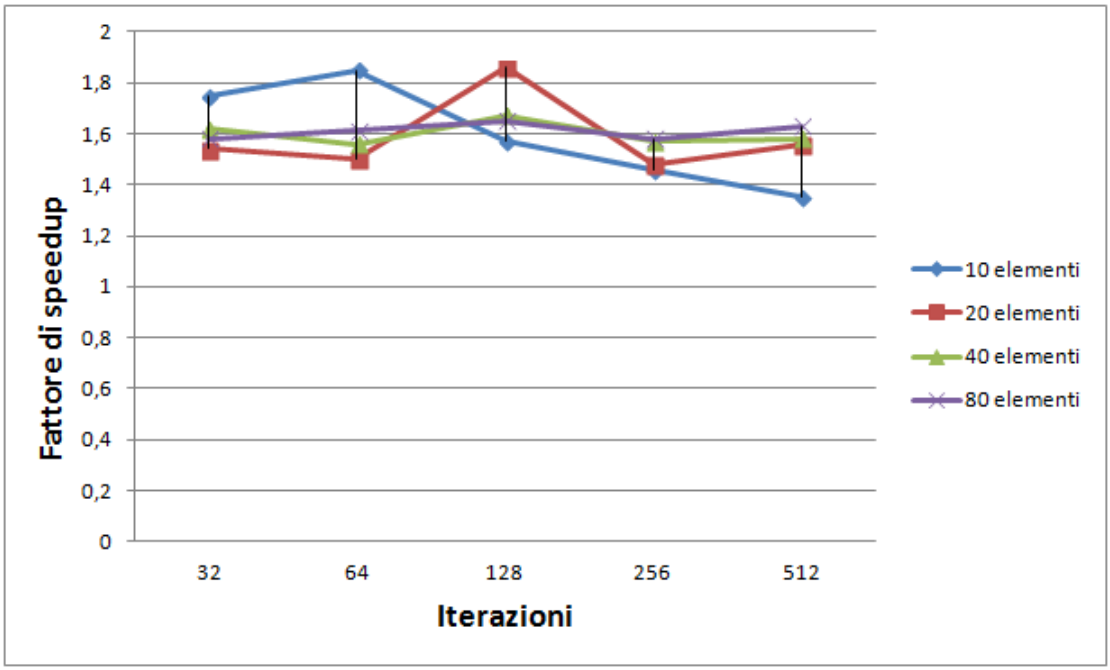


Figura 6.5: Fattore di speedup algoritmo parallelizzato su 2 thread

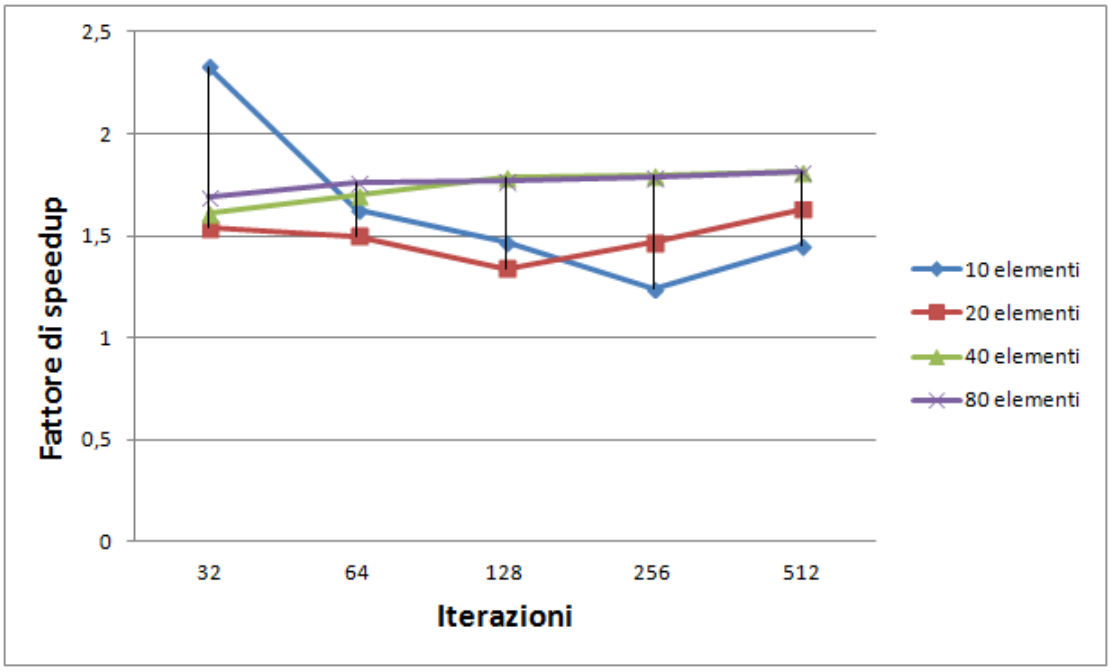


Figura 6.6: Fattore di speedup algoritmo parallelizzato su 4 thread

effettuare i cambi di contesto tende ad aumentare. Questo mostra intuitivamente che aumentare il numero di thread è sensato solo se aumenta anche la taglia del problema in ingresso.

Capitolo 7

Analisi dei risultati

Dopo aver analizzato la complessità temporale degli algoritmi sviluppati, in questa sezione verrà analizzata la bontà della soluzione trovata utilizzando GRASP.
Nelle tabelle seguenti con taglia si intende il numero di lamierini presenti nel semilavorato, con costo greedy il costo della soluzione greedy (in questo caso con soluzione greedy si intende una soluzione che sceglie sempre il migliore elemento localmente, senza nessuna randomizzazione); con costo grasp si intende il costo ottenuto con l'applicazione di grasp.
Nell'analisi sono riportate solo le tabelle relative ad esecuzioni con configurazioni di parametri che hanno fornito i risultati migliori.

Taglia	Costo Greedy	Costo GRASP	Rapporto GRASP/Greedy
10	391	382	0,97
11	530	508	0,96
12	859	661	0,77
13	904	733	0,81
14	893	786	0,88
15	893	768	0,86
16	944	962	1,02
17	1297	1142	0,88
18	1346	1150	0,85
19	1230	1230	1
20	1389	1168	0,84

Tabella 7.1: Prestazioni di GRASP. Parametri: lunghezza massima rcl=3, K=3, iterazioni=100

Taglia	Costo Greedy	Costo GRASP	Rapporto GRASP/Greedy
10	391	376	0,96
11	530	530	1
12	859	651	0,76
13	904	715	0,79
14	893	833	0,93
15	893	768	0,86
16	944	944	1
17	1297	1176	0,91
18	1346	1366	1,011
19	1230	1230	1
20	1389	1429	1,02

Tabella 7.2: Prestazioni di GRASP. Parametri: lunghezza massima rcl=3, K=5, iterazioni=100

Taglia	Costo Greedy	Costo GRASP	Rapporto GRASP/Greedy
10	391	385	0,98
11	530	501	,94
12	859	643	0,75
13	904	747	0,83
14	893	768	0,86
15	893	872	0,98
16	944	922	0,98
17	1297	1153	0,89
18	1346	1344	0,99
19	1230	1230	1
20	1389	1255	0,9

Tabella 7.3: Prestazioni di GRASP. Parametri: lunghezza massima rcl=5, K=5, iterazioni=100

Dall'analisi delle tabelle sono emersi i seguenti punti:

- Può accadere che la soluzione ottenuta mediante GRASP sia peggiore della soluzione greedy;
- Il fattore di miglioramento introdotto da GRASP dipende fortemente dalla morfologia della particolare istanza del problema (infatti dalle tabelle si nota che i casi in cui GRASP è poco efficiente sono all'incirca gli stessi per ogni settaggio dell'algoritmo).

Alla luce di quanto verificato, si è notato che un primo miglioramento dell'algoritmo base è quello di affiancare l'algoritmo greedy a GRASP e di utilizzare la soluzione migliore tra le due. La procedura (chiamata *First Enhancement Grasp*) è la seguente:

First Enhancement Grasp(iterations,K,α,Sheets,Pieces)
$C_1 \leftarrow$ *Greedy(Sheets,Pieces)*
$C_2 \leftarrow$ *Grasp(iterations,K,RCLsize,Sheets,Pieces)*
$K_1 \leftarrow Cost(C_1)$
$K_2 \leftarrow Cost(C_2)$
if $K_1 > K_2$
return C_2
else
return C_2

Il grafico seguente compara i risultati ottenuti nei tre casi. Si evidenzia che i migliori risultati mediamente sono quelli ottenuti con K pari a 3 e lunghezza della RCL pari a 3.

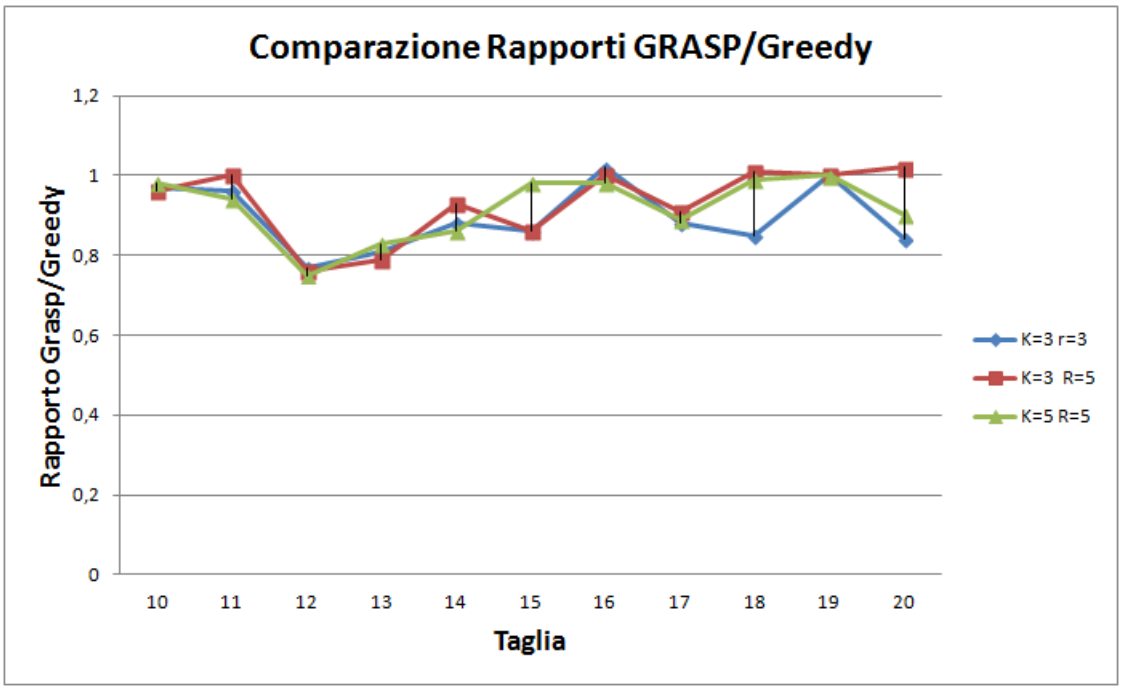


Figura 7.1: Confronto delle prestazioni ottenute con i tre settaggi analizzati in tabella. Chiaramente, minore è il rapporto ottenuto, migliore è la qualità della soluzione GRASP rispetto a quella ottenuta con l'algoritmo greedy. Si possono inoltre vedere alcuni valori maggiori di 1: in tali casi l'algoritmo GRASP ha portato a un risultato peggiore della soluzione greedy.

Capitolo 8

Applicativo EuclidSheets

Questa parte vuole fornire alcuni cenni sul software prodotto, ma invece che focalizzarsi sull'aspetto algoritmico come le sezioni precedenti si propone di mostrare il *front-end* dell'applicativo creato, cioè l'aspetto dell'applicazione lato user. La GUI *(Graphical user interface)* è stata creata interamente con i *Windows Forms* di Visual Studio 2012. Tale scelta è stata ispirata da diversi fattori:

- Il software deve essere disponibile per piattaforma Windows;
- I *Windows Forms* uniscono semplicità di utilizzo per lo sviluppatore a un aspetto finale dell'applicazione molto gradevole;
- Ultimo, ma non meno importante, le politiche dell'azienda presso la quale si è sviluppata l'applicazione.

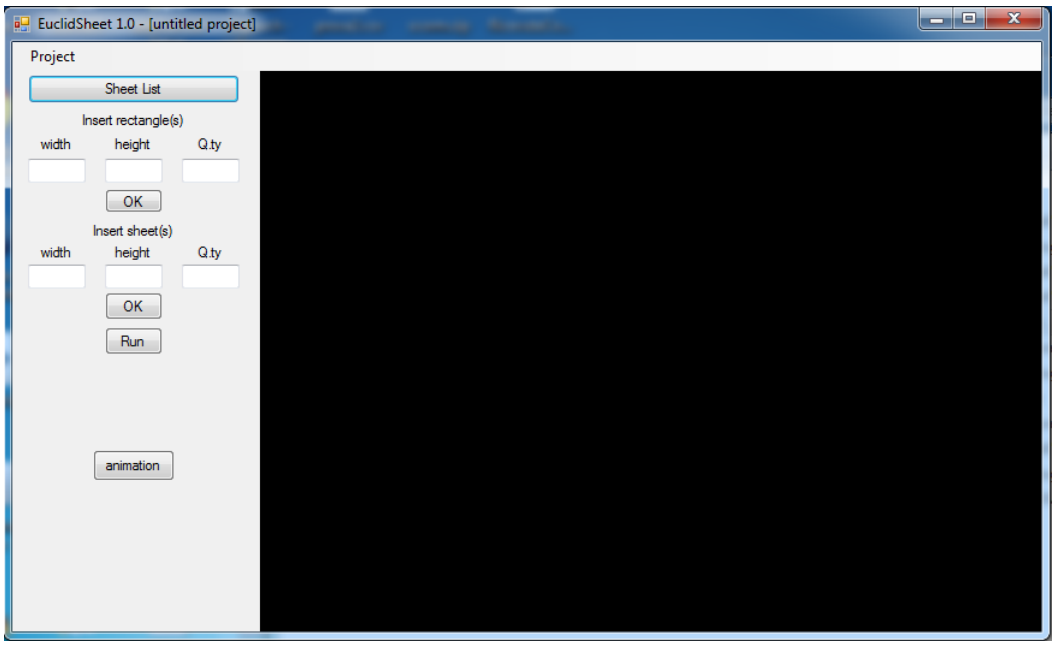


Figura 8.1: Interfaccia utente all'avvio. Si possono notare, sulla sinistra, i vari form e bottoni che consentono di inserire dati e di lanciare i vari algoritmi. La parte destra (di colore nero), contiene un visualizzatore molto simile a quello di applicativi quali AutoCAD. Qui verrà visualizzato il nesting calcolato e la sequenza di taglio.

8.1 Interfaccia utente

All'avvio viene visualizzata l'interfaccia principale dell'applicazione. Essa consente di inserire dati per un problema di nesting/cutting, di eseguire uno degli algoritmi precedenti e di salvare istanze di problemi in formato *.csv*.

Nell'interfaccia sono presenti i seguenti comandi:

- **Gruppo di form per inserimento lamierino**: tre form in cui si può inserire lunghezza, altezza e cardinalità di un lamierino. Per l'altezza e la lunghezza si possono inserire valori interi o decimali, mentre la cardinalità deve (banalmente) essere un numero intero.

- **Gruppo di form per inserimento foglio**: analogo al precedente form, questo gruppo si occupa dell'inserimento di una nuova lamiera.

- **Tasto visualizzazione tabellare**: Premendo questo tasto si apre una schermata formata da due tabelle riepilogative, una per i lamierini ed una per i fogli di lamiera. Le due tabelle sono modificabili dall'utente che può aggiungere o rimuovere righe.

- **Tasto Run**: il tasto *Run* lancia l'esecuzione dell'algoritmo. Una volta premuto appariranno dei nuovi pulsanti che potranno essere utilizzati dall'utente per visualizzare sulla schermata di destra l'ordine temporale dei singoli tagli.

- **Tasto Animation**: anche il tasto *Animation*, come il precedente, avvia l'esecuzione dell'algoritmo. In questo caso però non è l'utente a scorrere avanti ed indietro la sequenza di navigazione, ma i tagli successivi vengono evidenziati automaticamente uno ogni 500 millisecondi.

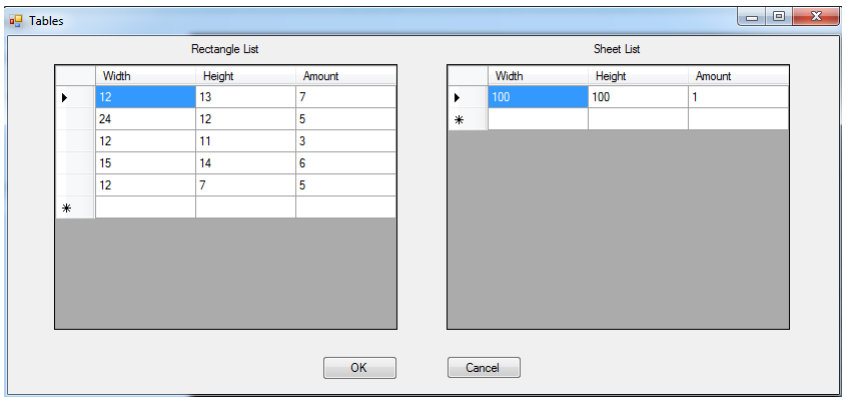


Figura 8.2: La schermata riepilogativa che appare all'utente premendo il tasto *Sheet List*: a sinistra vengono visualizzati i lamierini da ottenere mentre a destra si evidenziano i fogli.

8.2 Salvataggio e Apertura file

Il salvataggio di un file consente di salvare i lamierini ed i fogli di una particolare configurazione all'interno di un file dall'estensione *.csv*, sigla che sta ad indicare file di tipo *Comma Separated Value*, che tradotto in italiano significa "valori separati da virgola". Questo formato del tutto open[1] consente di creare file tabulari separando le colonne con una virgola e le righe andando a capo all'interno del file. Uno degli innegabili vantaggi derivanti dall'utilizzo di questo formato è la sua portabilità, in quanto esistono moltissimi parser e visualizzatori di file csv, sia liberi che commerciali. Si consideri ad esempio il file csv contenente i seguenti caratteri:

```
Nome,Cognome,Età
Antonio,Rossi,21
Marco,Bianchi,23
```

Esso, se letto da un visualizzatore di CSV creerà una tabella del tipo:

Nome	Cognome	Età
Antonio	Rossi	21
Marco	Bianchi	23

Nel caso di EuclidSheets, il file csv prodotto contemrrà i seguenti campi:

- **Type**: indica se l'oggetto è un lamierino (valore impostato a 0) o un foglio (valore impostato ad 1);
- **Width**: larghezza del componente;
- **Height**: altezza del componente;
- **Amount**: cardinalità del componente.

Con questa formattazione dei dati è possibile salvare i lamierini ed i fogli all'interno di un solo file csv.

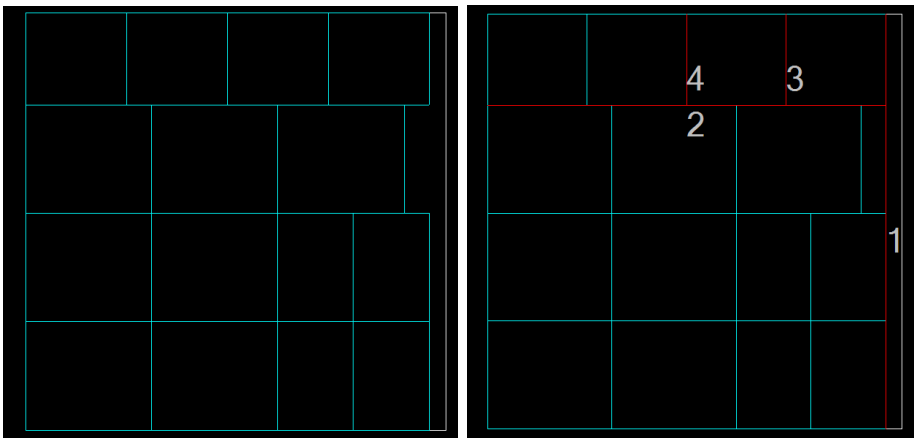
8.3 Algoritmo di lettura e scrittura su file

L'applicazione possiede due strutture dati (Liste) al suo interno: una per gestire i lamierini ed una per i fogli. All'apertura di un file di configurazione csv, un parser scorre il file stesso e si occupa di popolare le liste. Le due liste vengono aggiornate ogni volta che un oggetto viene aggiunto dall'utente. Al momento di salvare, viene istanziato un oggetto che si occupa della creazione di un nuovo file, raccoglie tutti gli oggetti contenuti nelle due liste e li salva sul file stesso.

[1] open, o aperto: significa che la sua codifica è uno standard pubblicato e disponibile a tutti.

8.4 Visualizzazione sequenza di taglio

Una volta inseriti i dati desiderati, con la pressione del tasto **Run** esegue l'algoritmo di disposizione dei lamierini sui fogli di lamiera, seguito dall'algoritmo di ottimizzazione della sequenza di taglio. I risultati vengono mostrati nel riquadro di destra della finestra principale. Tale riquadro è un componente aggiuntivo di Windows Forms creato da Euclid Labs, e ha la funzione di visualizzare oggetti grafici bidimensionali. Il componente è già provvisto di molte funzioni di base, quali zoom, navigazione, e possibilità di catturate eventi quali click del mouse e pressioni di tasti. Una volta completata l'esecuzione dell'algoritmo sul componente appaiono i semilavorati, al cui interno sono disposti i lamierini. I segmenti che rappresentano gli oggetti sono di colore azzurro chiaro. A questo punto, mediante il pannello di navigazione è possibile visualizzare la sequenza dei tagli. Ad ogni pressione sul tasto **Next** apparirà un segmento di colore rosso: tale segmento rappresenta un taglio. In corrispondenza del taglio è presente anche un numero intero, che indica la posizione del taglio all'intero della sequenza.



(a) Stato Iniziale (b) Dopo 4 Tagli

Figura 8.3: Rappresentazione del foglio nell'applicazione. La prima immagine mostra lo stato iniziale, senza nessun taglio evidenziato i rosso. La seconda figura mostra invece i primi 4 tagli effettuati. Ogni taglio viene etichettato secondo il suo ordine nella sequenza

Capitolo 9

Miglioramenti Futuri

Considerando i risultati ottenuti sia in termini di complessità computazionale sia in termini di qualità delle soluzioni, in questo capitolo si intende fornire un accenno a possibili percorsi da intraprendere per ulteriori miglioramenti futuri.

Il primo punto sul quale è senz'altro possibile effettuare dei raffinamenti è il modello di costo utilizzato per valutare la bontà dei tagli. La difficoltà che sta alla base di questo punto è data dal fatto che il modello adottato ha cercato di sintetizzare opinioni basate su valutazioni empiriche fornite da esperti, in quanto il problema è di carattere pratico e non esiste una formulazione matematica esatta.

La versione di GRASP utilizzata è una variante della ricerca locale classica. Gli algoritmi metaeuristici sono basati su due prinicipi: *exploration* (ricerca di nuove soluzioni), ed *exploitation* (sfruttamento delle soluzioni già trovate per cercarne dei miglioramenti). La metaeuristica GRASP pura (come la local search) penalizza fortemente la fase di exploitation a scapito della fase di exploration. Un possibile miglioramento può essere ottenuto combinando GRASP con l'euristica *path relinking* [32].

L'euristica path-relinking considera la soluzione migliore finora incontrata nello svolgimento di GRASP (chiamata *elite solution* x_{elite}). Una volta calcolata una nuova soluzione x_n, si calcola il numero di modifiche necessarie per arrivare da x_{elite} fino ad x_n. Se una di queste soluzioni intermedie $x_{i1}, x_{i2}, ..., x_{ik}$ ha costo minore di x_{elite}, essa diventa la nuova x_elite.

Il problema nell'applicazione di path-relinking risiede nel fatto che non sempre è così semplice calcolare la sequenza di passaggi che portano da una soluzione ad un'altra.

Parlando dei possibili miglioramenti dal punto di vista della velocità di esecuzione dell'algoritmo, si evidenzia come il calcolo della soluzione greedy (di costo cubico al caso peggiore), rappresenti un grosso handicap per le prestazioni. Sviluppando un algoritmo più veloce per la detection dei tagli si otterrebbe un deciso incremento delle prestazioni.

Un ulteriore miglioramento alle prestazioni potrebbe essere fornito creando una versione di GRASP ibrida, in quanto nell'analisi delle prestazioni si è evi-

denziato come cambia il rendimento del codice parallelo a seconda della taglia dell'istanza.

Conclusioni

Il lavoro svolto mette in evidenza come grazie all'applicazione di GRASP si riesca ad ottenere un netto miglioramento della qualità della soluzione trovata con un algoritmo di tipo greedy. Inoltre la metaeuristica si è dimostrata scalabile e facilmente parallelizzabile.

Il punto critico dell'algoritmo sviluppato è la fase di ricerca dei tagli ammissibili, che pregiudica la velocità di esecuzione dell'intero programma. L'aumento della velocità di esecuzione dell'algoritmo greedy consentirebbe una maggiore scalabilità.

Altro punto critico per lo sviluppo di software dalle buone prestazioni è la creazione di un modello di costo che riesca ad essere il più possibile aderente alle considerazioni empiriche elaborate nel Capitolo 3.

Nel complesso, la velocità di esecuzione resta comunque apprezzabile così come la qualità delle soluzioni trovate; la taglie ridotte delle applicazioni pratiche (nell'ordine di 100 elementi), garantiscono delle buone performance.

Appendice A

Cenni sugli algoritmi metaeuristici

Dato che nell'intero svolgimento del lavoro si fa molto spesso riferimento agli algoritmi metaeuristici, si è deciso di introdurre questa sezione con lo scopo di fornire alcuni cenni teorici di base su di essi.

Gli algoritmi metaeuristici sono una branca dell'intelligenza artificiale che affronta problemi di ottimizzazione (problemi di massimo o minimo).
Sono molto utilizzati quando il problema da risolvere è NP-Hard o comunque quando si opera in ambiti in cui è importante la velocità di esecuzione (multimedia, sistemi embedded con risorse limitate). Il termine metaeuristico si deve al fatto che il loro funzionamento molto spesso è basato su fenomeni fisici o sul comportamento degli animali (gravità, moto di particelle, colonie di formiche).
Questi algoritmi, tuttavia, non garantiscono l'ottenimento del valore ottimo, anche se con buoni settaggi molto spesso riescono ad ottenere risultati approssimati che si discostano di poco dall'ottimo stesso.

Caratteristiche comuni degli algoritmi metaeuristici

Gli algoritmi che verranno esaminati in seguito hanno delle caratteristiche comuni:

- sono di carattere iterativo;
- sono facilmente parallelizzabili;
- si basano su due principi: *exploration*, cioè l'esplorazione del dominio per la ricerca di nuove soluzioni, ed *exploitation*, cioè lo sfruttamento delle soluzioni più promettenti;
- per garantirne un buon funzionamento è richiesto un lavoro di tuning sui parametri, che molto spesso è il vero scoglio per la buona riuscita di un algoritmo metaeuristico.

Nelle sezioni seguenti si descriveranno brevemente gli algoritmi metaeuristici più conosciuti ed utilizzati.

Local Search

Algoritmo che funziona bene su problemi su dominio discreto, local search è una ricerca iterativa che avviene mediante creazioni di intorni della soluzione iniziale. Si supponga di dover risolvere un problema di massimo. L'algoritmo parte da una soluzione casuale e ne crea una neighbourhood (cioè un insieme di soluzioni leggermente diverse); se una di queste soluzioni ha valore maggiore della soluzione precedente viene scelta come soluzione di partenza per l'iterazione successiva.
L'algoritmo termina l'esecuzione quando si esauriscono le iterazioni o quando la neighbourhood non presenta soluzioni migliori di quella attuale.
Questo algoritmo presenta dei vantaggi innegabili:

- Pochi parametri su cui fare il tuning;
- Semplicità di implementazione;
- Velocità di esecuzione.

L'algoritmo presenta però un conveniente non trascurabile: se nella ricerca delle soluzioni si incontra un punto di massimo locale, termina in quanto non è più in grado di migliorare la soluzione trovata.
Un miglioramento che cerca di ovviare a questo inconveniente viene introdotto nella *Multistart Local Search*. Come suggerisce il nome, si lanciano diverse local search utilizzando diverse soluzioni iniziali nella speranza che uno dei punti di massimo locale trovati sia il punto di massimo del problema.

In letteratura si trovano diversi esempi di applicazioni di local search, soprattutto su problemi non polinomiali, ad esempio:

- Vertex cover problem;
- Traveling salesman problem;
- Boolean satsfiability problem.

Tabu Search

Tabu Search, inventato da Fred Glover [17], [18], cerca di superare il limite di Local Search, l'arresto sui punti di minimo locali.
L'algoritmo funziona in modo del tutto simile, anche se in questo caso sono consentite scelte che portino a "soluzioni peggioranti". Tuttavia, così facendo, si rischia di ritornare nel punto di minimo locale dopo poche iterazioni. Per evitare ciò l'algoritmo associa alla funzione di ricerca una tabella che tiene in memoria alcune informazioni relative al percorso dell'algoritmo all'interno del dominio delle soluzioni, consentendo quindi di impedire di ricadere in un punto di minimo locale (introduzione delle mosse *Tabù*).

La tabelle introducono della memoria nell'algoritmo, che può così ricordare il percorso fatto evitando di ricadere in cicli. Tale memoria può essere di tre tipi:

- *Memoria a breve termine*: la lista delle soluzioni visitate recentemente. Se una soluzione potenziale è presente in tabella, non può essere rivisitata prima di un certo numero di iterazioni (*expiration point*).
- *Memoria a lungo termine*: Un elenco di norme volte a influenzare la ricerca verso i settori più promettenti dello spazio di ricerca.
- *Memoria a lungo termine*: Regole che cercano di fornire diversità nello spazio di ricerca (exploration).

L'algoritmo presenta però due inconvenienti:

- funziona solo su domini discreti;
- tende a concentrarsi su di uno spazio ristretto all'interno del dominio, pregiudicando la bontà della soluzione trovata nel caso il dominio sia molto vasto (ad esempio in domini con numero di dimensioni molto alto).

Simulated Annealing

Questo algoritmo si basa sul fenomeno fisico della tempra dei metalli.
La tempra ha lo scopo di portare i reticoli cristallini al punto di energia minima; analogamente l'algoritmo ha lo scopo di trovare un punto di minimo assoluto in presenza di minimi locali.
Partendo da uno *stato iniziale* arbitrario l'algoritmo cerca di arrivare ad un punto ad *energia minima*.

Ad ogni iterazione dell'algoritmo, viene deciso arbitrariamente se trasferirsi dallo stato corrente s ad uno stato adiacente s^*. Questa operazione viene ripetuta finché lo stato corrente non ha energia $E(s)$ minore della soglia richiesta dall'applicazione, oppure non si raggiunge il limite massimo di iterazioni impostato dall'utente.

La probabilità di transizione da uno stato all'altro è data dalla *acceptance probability function* $P(e, e^*, T)$, con $e = E(s)$, $e^* = E(s^*)$, ed un parametro che varia nel tempo T che è detto *temperatura*.
Con il diminuire di T, $P(e, e^*, T)$ tende a zero se $e^* > e$ e ad uno se $e^* < e$. Questo significa che al diminuire di T il sistema tende a muoversi verso valori che ne minimizzino l'energia, (*downhill move*). Analogamente le transizioni verso soluzioni ad energia superiore vengono chiamate *uphill moves*.

Al crescere di T, aumenta la probabilità che il sistema esegua una uphill move. Queste transizioni verso stati ad energia superiore servono ad evitare che l'algoritmo si blocchi su punti di minimo locale.

La *acceptance probability function* viene solitamente costruita in modo che la probabilità di una uphill move decresca al crescere del valore di $|e^* - e|$, così da favore le uphill move che incrementino di poco l'energia del sistema.

Per applicare con successo SA devono essere scelti i seguenti parametri:

- la funzione energia $E(s)$;
- la funzione di creazione dell'intorno delle soluzioni $neighbourhood(S)$;
- la acceptance probability function $P(s, s^*, T)$;
- la funzione che regola l'andamento della temperatura al variare del tempo x, $T(x)$;
- il valore iniziale della temperatura t_0.

Non esistono scelte di tali parametri che siano valide per tutti i problemi. Ogni problema approcciato con SA richiede una fase di settaggio dei parametri "ad hoc".

Ulteriori approfondimenti su *Simulated Annealing* possono essere trovati in [25], [6], [22].

A.1 Ant Colony Optimization

ACO è usato per risolvere problemi riconducibili a percorsi su grafi. Esso è basato sul comportamento delle colonie di formiche alla ricerca di cibo. La scoperta dell'algoritmo si deve a Marco Dorigo [7] [10].

Nella realtà le formiche si muovono inizialmente lungo un percorso casuale. Una volta trovato del cibo esse riescono a trovare il loro nido seguendo le tracce odorose di feromone che secernono al loro passaggio.
Le formiche, se si imbattono nella traccia di feromone lasciata da altre formiche, la seguono, secernendo a loro volta altro feromone che non fa altro che rinforzare la traccia.
Dopo un certo lasso di tempo, il percorso inizialmente casuale delle formiche si stabilizza lungo un unico percorso che conduce dal cibo al nido.
Il feromone nel tempo evapora. Questo implica che i percorsi più lunghi hanno un potere attrattivo minore rispetto ai percorsi brevi, in quanto la quantità di feromone è minore.

Nell'algoritmo, una formica è vista come un agente. L'agente costruisce iterativamente delle soluzioni del problema partendo da un nodo iniziale del grafo ed arrivando un nodo goal. Ad ogni iterazione dell'algoritmo, la formica attraversa un lato del grafo, spostandosi da un nodo ad un altro ad esso collegato.
Dato il nodo corrente ed un insieme di possibili nodi adiacenti, la formica ha una

certa probabilità di trasferirsi in uno qualsiasi di questi nodi.
Se il nodo x rappresenta il nodo corrente e Z l'insieme dei nodi raggiungibili da x, la probabilità p_{xy} di trasferirsi al nodo $y \in Z$ è data da:

$$p_{xy} = \frac{\left(\tau_{xy}^{\alpha}\right)\left(\eta_{xy}^{\beta}\right)}{\sum_{z \in Z}\left(\tau_{xz}^{\alpha}\right)\left(\eta_{xz}^{\beta}\right)}$$

Nella formula, τ_{xy} è la quantità di feromone depositato, $\alpha > 0$ è un parametro che indica quanto il feromone incide sulla scelta del percorso (tanto più α è grande tanto maggiore è l'influenza del feromone), η_{xy} indica la convenienza della transizione (in genere una funzione basata sul peso del lato del grafo), $\beta \geq 1$ è il parametro che influisce su η.

Quando tutte le formiche hanno completato un percorso (cioè sono giunte ad una soluzione), si procede all'aggiornamento della quantità di feromone per l'iterazione successiva. La nuova quantità di feromone su ogni lato del grafo all'iterazione i diventa:

$$\tau_{xy}(i) \leftarrow (1-\rho)\,\tau_{xy}(i-1) + \sum_{ant \in A} \tau_{xy}^{ant}(i-1)$$

Nella formula, $\tau_{xy}(i-1)$ rappresenta la quantità di feromone sul lato xy nell'iterazione (i-1)-esima prima del passaggio delle formiche, $\tau_{xy}^{ant}(i-1)$ rappresenta la quantità deposita dalla formica ant nell'iterazione (i-1)-esima, e $0 < \rho < 1$ rappresenta la velocità di evaporazione del feromone.

Ecco alcuni esempi di applicazione di ACO [36]:

- Problemi di scheduling:
  - Job-shop scheduling problem (JSP)
  - Open-shop scheduling problem (OSP)
  - Permutation flow shop problem (PFSP)
  - Single machine total tardiness problem (SMTTP)
  - Single machine total weighted tardiness problem (SMTWTP)
  - Resource-constrained project scheduling problem (RCPSP)[
  - Group-shop scheduling problem (GSP)
  - Single-machine total tardiness problem with sequence dependent setup times (SMTTPDST)
  - Multistage Flowshop Scheduling Problem (MFSP) with sequence dependent setup/changeover times
- Problemi di Vehicle Routing:
  - Capacitated vehicle routing problem (CVRP)

- Multi-depot vehicle routing problem (MDVRP)
- Period vehicle routing problem (PVRP)
- Split delivery vehicle routing problem (SDVRP)
- Stochastic vehicle routing problem (SVRP)
- Vehicle routing problem with pick-up and delivery (VRPPD)
- Vehicle routing problem with time windows (VRPTW)
- Time Dependent Vehicle Routing Problem with Time Windows (TD-VRPTW)
- Vehicle Routing Problem with Time Windows and Multiple Service Workers (VRPTWMS)

Bibliografia

[1] Renata M. Aiex and Mauricio G.C. Resende. Parallel strategies for grasp with path-relinking. *AT&T Labs Research Technical Report TD-5SQKM9.*, 2003.

[2] R.M. Aiex, S. Binato, and M.G.C. Resende. Parallel grasp with path relinking for job shop scheduling. *AT&T Labs Research Technical Report*, 2001.

[3] Joseph Albahari and Ben Albahari. *C# 5.0 in a Nutshell: The Definitive Reference*. O'Reilly, 2012.

[4] A. Bortfeldt, H. Gehring, and D. Mack. A parallel tabu search algorithm for solving the container loading problem. *Parallel Comput.*, 29(5):641–662, May 2003.

[5] E. K. Burke, A. J. Eckersley, B. McCollum, S. Petrovic, and R. Qu. Hybrid variable neighbourhood approaches to university exam timetabling. *European Journal of Operational Research, 2009, vol. 206, issue 1, pages 46-53*, 2006.

[6] V. Cerny. Thermodynamical approach to the traveling salesman problem: An efficient simulation algorithm. *Optimization Theory and Applications*, 45:41–51, 1985.

[7] A. Colorni, M. Dorigo, and V. Maniezzo. Distributed optimization by ant colonies. *actes de la premiÃ¨re confÃ©rence europÃ©enne sur la vie artificielle*, 1992. Published by Elsevier.

[8] S. Consoli, K. Darby-Dowman, N. Mladenovic, and J. A. Moreno Perez. Greedy randomized adaptive search and variable neighbourhood search for the minimum labelling spanning tree problem. *European Journal of Operational Research, 2009, vol. 196, issue 2, pages 440-449*, 2008.

[9] H. Dickhoff. A typology of cutting and packing problems. *European Journal Of Operational Research*, 1990.

[10] M. Dorigo. Optimization, learning and natural algorithms. *PhD Thesis*, 1992. Politecnico di Milano.

[11] Adam Freeman. *Pro .NET 4.0 Parallel Programming in C#*. Apress, 2010.

[12] H. Gehring and A. Bortfeldt. A genetic algorithm for solving the container loading problem. *International Transactions in Operational Research 4:401-418*, 1997.

[13] H. Gehring and A. Bortfeldt. A hybrid genetic algorithm for the container loading problem. *European Journal of Operational Research 131:143-161*, 2001.

[14] H. Gehring and A. Bortfeldt. A parallel genetic algorithm for solving the container loading problem. *International Transactions in Operational Research 9:497-511*, 2002.

[15] P.C. Gilmore and R.E. Gomory. A linear programming approach to the cutting-stock problems. *Operations Research 9: 849-859*, 1961.

[16] P.C. Gilmore and R.E. Gomory. A linear programming approach to the cutting-stock problems - part ii. *Operations Research 11: 863-888*, 1961.

[17] F. Glover. Tabu search - part 1. *ORSA Journal on Computing 1*, 1989.

[18] F. Glover. Tabu search - part 2. *ORSA Journal on Computing 2*, 1990.

[19] Fred Glover and Manuel Laguna. *Tabu Search*. Kluwer Academic Publishers, Norwell, MA, USA, 1997.

[20] Fred Glover and Claude McMillan. The general employee scheduling problem: an integration of ms and ai. *Comput. Oper. Res.*, 13(5):563–573, May 1986.

[21] Melissa D. Goodman, Kathryn A. Dowsland, and Jonathan M. Thompson. A grasp-knapsack hybrid for a nurse-scheduling problem. *Journal of Heuristics, January 2009; 15:351-379. pp.351-379*, 2009.

[22] V. Granville, M. Krivánek, and J. P. Rasson. Simulated annealing: A proof of convergence. *IEEE Trans. Pattern Anal. Mach. Intell.*, 16(6):652–656, June 1994.

[23] Gaston Hillar. *Professional Parallel Programming with C#: Master Parallel Extensions with .NET 4*. Wrox, 2010.

[24] `http://www.directindustry.com/`. Elenco costruttori di cesoie. Guillotine builders.

[25] S. Kirkpatrick, C. D. Gelatt, and M. P. Vecchi. Optimization by simulated annealing. *Science*, 220:671–680, 1983.

[26] S. L. Martins, M. G. C. Resende, C. C. Ribeiro, and P. M. Pardalos. A parallel grasp for the steiner tree problem in graphs using a hybrid local search strategy. *Journal of Global Optimization*, 2000.

[27] Christian Nagel, Bill Evjen, Jay Glynn, Karli Watson, and Morgan Skinner. *Professional C# 2012 and .NET 4.5*. Wrox, 2012.

[28] L.S. Ochi, L.S. Vianna, M.B. da Silva, and M. de Assumpcao Dummond. Sequential and parallel metaheuristics based on grasp and vns for solving the traveling purchaser problem. *Fourth Metaheuristics International Conference*, 2001.

[29] P. M. Pardalos, L. Pitsoulis, and M. G. C. Resende. A parallel grasp for max-sat problems. *Applied Parallel Computing Industrial Computation and Optimization Lecture Notes in Computer Science Volume 1184, 1996, pp 575-585*, 1996.

[30] Devi Priya, Joshua Samuel Raj, and V. Vasudeven. Article: Hybridized tabubfo algorithmin grid scheduling. *International Journal of Computer Applications*, 63(11):43–47, February 2013. Published by Foundation of Computer Science, New York, USA.

[31] Mauricio G.C. Resende and Celso C. Ribeiro. Greedy adaptive search procedures. *AT&T Labs Research Technical Report TD-53RSJY, version 2*, 2002.

[32] M.G.C. Resende and C.C. Ribeiro. Grasp with path relinking: recent advances and applications. *AT&T Labs Research Technical Report TD-5TU726*, 2003.

[33] Luis Santos, Daniel Madeira, Esteban Clua, Simone Martins, and Alexandre Plastino. A parallel grasp resolution for a gpu architecture. *International conference on metaheuristics and nature inspired computing*, 2010.

[34] Guntram Scheithauer. Algorithms for the container loading problem. In *Operations Research Proceedings 1991, Springer-Verlag*, pages 445–452. Springer-Verlag, 1992.

[35] Andrew Troelsen. *Pro C# and the .NET 4.5 Framework 6th Edition*. Apress, 2012.

[36] Wikipedia. Ant colony optimization. `http://en.wikipedia.org/wiki/Ant_colony_optimization_algorithms#Example_pseudo-code_and_formulae`.

[37] Wikipedia. Cesoia (meccanica). `http://it.wikipedia.org/wiki/Cesoia_(meccanica)`.

Ringraziamenti

Desidero ringraziare in primo luogo tutto lo staff di Euclid Labs s.r.l. per il supporto fornitomi durante lo svolgimento della tesi ed in particolare Roberto Polesel, Walter Zanette e Matteo Garofalo. Un ringraziamento va anche all' Ing. Francesco Sambo, per avermi fornito utili spunti sugli algoritmi metaeuristici e per la sua cordiale disponibilità. Voglio inoltre ringraziare il Dott. Fabio Bottero per i consigli disinteressati e la gentilezza dimostrata negli anni di corso passati insieme. Un ringraziamento va al Dott. Andrea Bisson per avermi aiutato nella redazione del template in LaTeXper la stesura di questo lavoro. Ringrazio inoltre il Dott. Giada Quagliato per avermi supportato e soprattutto sopportato in questi anni di studi. Ultimo ma non meno importante, ringrazio i miei zii Anna Ferini e Severino Fontolan per avermi messo a disposizione il computer con cui sviluppare e testare tutto il codice necessario alla stesura di questo lavoro.