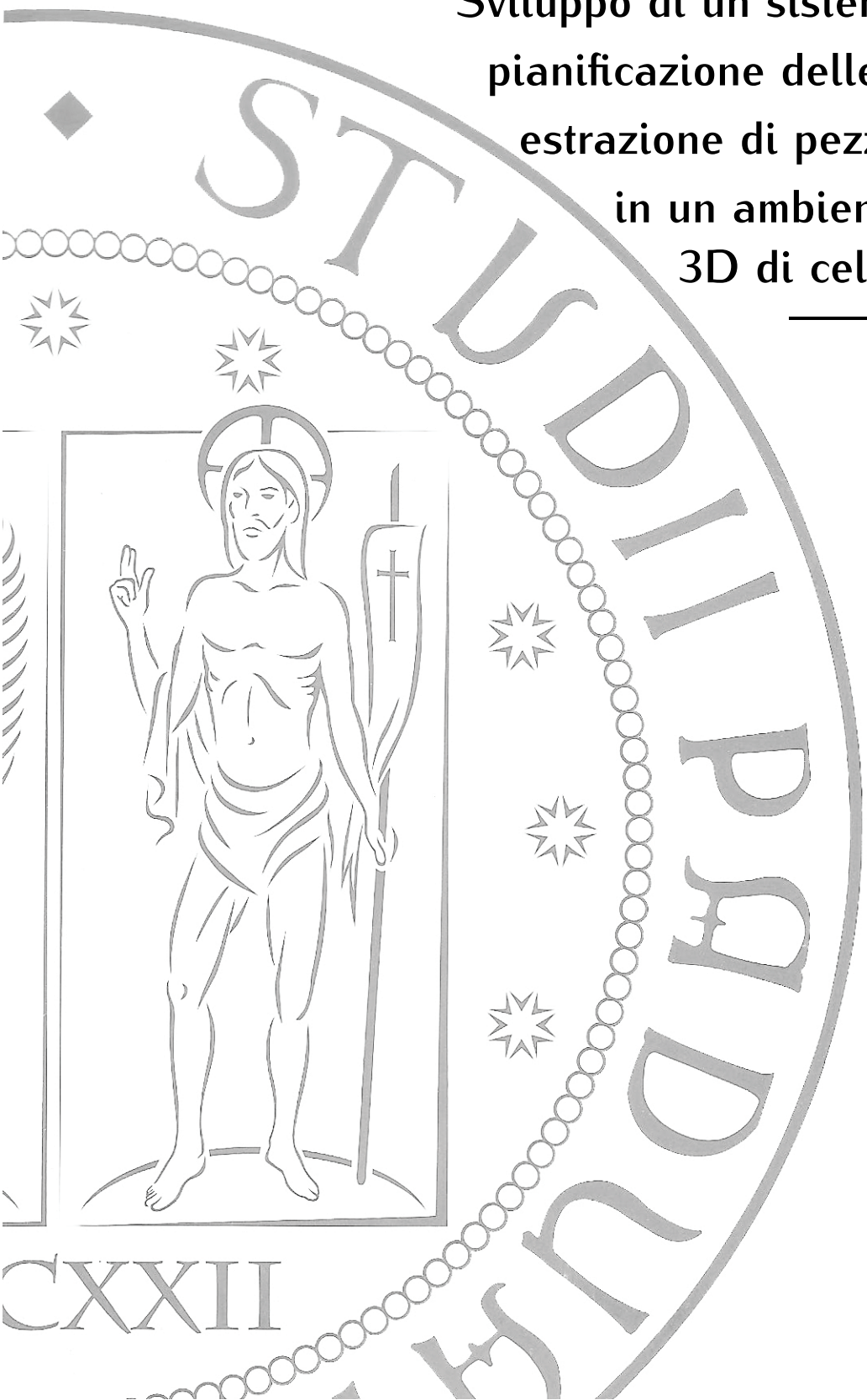

DEPARTMENT OF INFORMATION ENGINEERING
UNIVERSITY OF PADOVA

Corso di Laurea in Ingegneria Informatica

Tesi di Laurea magistrale

Sviluppo di un sistema software per la pianificazione delle operazioni di estrazione di pezzi da una pressa in un ambiente di simulazone 3D di celle robotizzate

Candidato:
Alessandro Zanella

Relatore:
Prof. Enrico Pagello

Correlatore:
Stefano Tonello

12 Marzo 2013
A.A. 2012/2013

A mio padre che mi ha fatto capire
quel che è veramente importante nella vita

Indice

Elenco delle figure

Capitolo 1

Introduzione

Lo scopo di questa tesi è lo studio e lo sviluppo di un algoritmo di pianificazione per l'estrazione di pannelli di lamiera da una pressa tramite un robot manipolatore. Durante il processo di piegatura di pannelli di lamiera accade che il pannello possa prendere una forma per cui risulta che l'estrazione del lavorato dalla pressa sia un'operazione complessa. Questo, di conseguenza, rende difficoltosa la programmazione del robot. Per riuscire a identificare un software funzionante, è stato necessario studiare e poi sviluppare un tipo di algoritmo dedicato appositamente al problema proposto, che lo risolva in un tempo ragionevole e che trovi una soluzione il più vicina possibile a quella ottimale. Il software realizzato è parte del software di simulazione di processo di piegatura di pannelli di lamiera attraverso celle industriali robotizzate SimulEasy2 della IT-Robotics, presso la quale è stata realizzata la tesi. Il software è stato realizzato in C++ utilizzando le librerie messe a disposizione dalla IT-Robotics. Durante lo sviluppo della tesi, oltre a realizzare un algoritmo di planning funzionante per il problema dell'estrazione di fogli di lamiera, si è indagato se un tipo di algoritmo simile possa essere utilizzato per risolvere il problema del *assembly planning*.

1.1 Struttura della tesi

Di seguito sono elencati i contenuti dei capitoli successivi:

- **descrizione del problema**: in questo capitolo è descritto il processo di piegatura dei pannelli di lamiera e quindi il problema affrontato. Il capitolo è suddiviso in 8 sezioni:
  - 2.1 sono descritte quali funzionalità deve avere un simulatore di cella industriale e quale sia il suo scopo;
  - 2.2 è descritto come avviene il processo di piega;
  - 2.4 è descritto come sono rappresentati gli oggetti tridimensionali nel simulatore;
  - 2.5 è descritto in modo informale come è necessario affrontare il problema;
  - 2.6 sono descritte le operazioni disponibili nel software per muovere il robot;
  - 2.7 è spiegato perché questa situazione necessita di un algoritmo di pianificazione particolare;
  - 2.8 sono descritti gli algoritmi già presenti in SimulEasy2.
- **algorimi di pianificazione**, in questo capitolo è definito formalmente il problema di pianificazione e sono spiegati due algoritmi di pianificazione che sono stati alla radice della soluzione trovata. Il capitolo è diviso in 4 sezioni:
  - 3.1 descrizione formale del problema;
  - 3.2 breve spiegazione generale del problema;
  - 3.3 descrizione dell'algoritmo ARW;
  - 3.4 descrizione degli algoritmi di pianificazione basati su alberi RRT;

- **Realizzazione algoritmo**, in questo capitolo sono riportati tutti i tentativi effettuati per realizzare il programma e descritto, infine,l'algoritmo creato: Il capitolo è suddiviso in 7 sezioni:
  - 4.2 sono descritti i vari tentativi realizzati;
  - 4.3 è riportata la struttura dell'algoritmo;
  - 4.4 è spiegata la funzione `ChoosePosition`;
  - 4.5 descrizione della funzione `ExtractSimple`;
  - 4.6 descrizione della funzione `ExeOperation`;
  - 4.6.7 descrizione della funzione `AdjustMetrics`;
  - 4.7 è discusso brevemente l'algoritmo nella sua completezza.
- **Risultati** in questo capitolo sono riportati i risultati ottenuti e le possibili varianti. Il capitolo è diviso in 2 sezioni:
  - 5.1 sono descritti i risultati ottenuti, spiegando i pregi e i difetti dell'algoritmo;
  - 5.2 è riportato un caso specifico di cui il programma riesce a trovare la soluzione.
- **Conclusione** il capitolo presenta le conclusioni finale ed una piccola sintesi del lavoro svolto.

Capitolo 2

Descrizione del problema

2.1 Simulatore del processo di piega

Il processo di piegatura di lamiere, oggi, può essere reso completamente automatico utilizzando un robot manipolatore, il quale si occupa in modo del tutto automatico di caricare e scaricare i pezzi dentro la pressa. La programmazione del robot e degli altri macchinari è un processo particolarmente oneroso in termini di tempo e la presenza di errori nel programma potrebbe comportare collisione tra i diversi elementi della cella, danneggiando, quindi, gli stessi.

Per risolvere questi problemi sono stati creati dei simulatori delle celle industriali per assistere gli operatori nella creazione dei programmi per i robot. L'operatore può provare a realizzare e simulare l'operazione di piega scritta in un ambiente 3D del tutto simile alla cella automatica reale. Oltre a questo il simulatore può generare direttamente il codice robot evitando quindi la scrittura diretta del programma. Il simulatore per essere efficace deve, inoltre, fornire tutti gli strumenti per impostare e creare una cella industriale, modificando i diversi elementi, come il tipo di robot e macchinari presenti. In sintesi il simulatore deve garantire le seguenti funzionalità:

- **creazione delle cella di lavoro**, tramite la quale è possibile importare i modelli 3D dei diversi macchinari, impostare proprietà fisiche e

quindi creare una cella personalizzata, la quale dovrà essere simile il più possibile alla cella reale;

- **simulazione del processo di piegatura della lamiera**, tramite la quale è possibile simulare il processo di piega, potendo generare in modo automatico l'intero processo di piega e modificare ogni singola operazione.

In figura 2.1 è mostrata una schermata del simulatore SimulEasy2, nella quale è possibile vedere la rappresentazione tridimensionale della cella di lavoro.

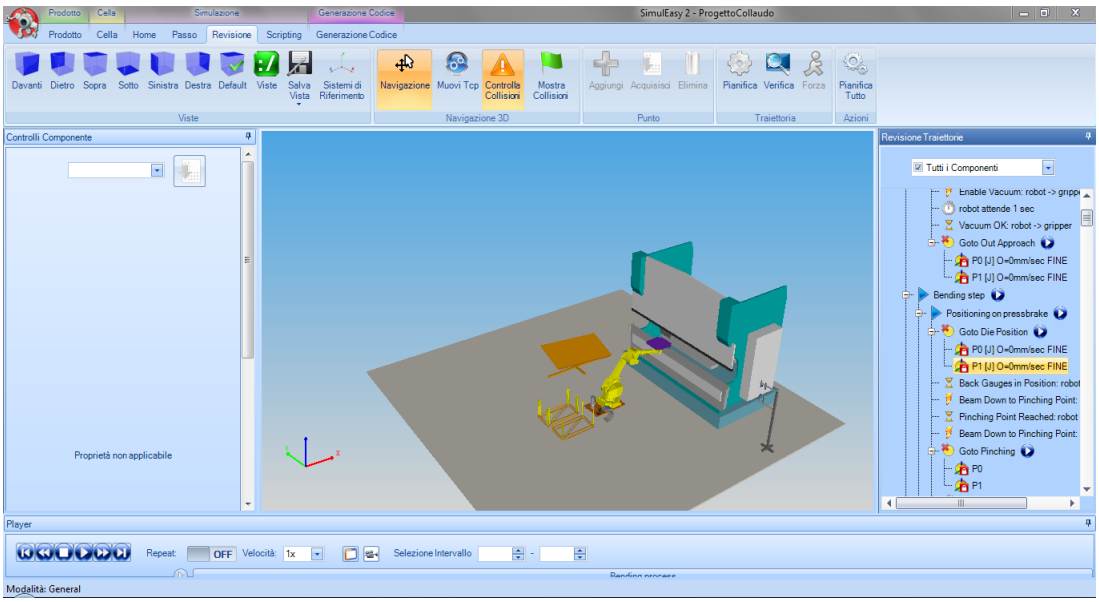

Figura 2.1: Schermata del simulatore SimulEasy2

2.2 Descrizione del processo di piega

In questa sezione è descritto come avviene il processo di piegatura. I pannelli di lamiera sono piegati comprimendo il pezzo fra due oggetti fino ad ottenere la posizione voluta. Il pezzo è posto su un supporto, detto **matrice**, che presenta una cavità. Uno strumento, detto **punzone**, è premuto contro

la lamiera sopra la matrice, fino ad ottenere la piega voluta. La piega è determinata dal tipo di matrice e dal tipo di punzone utilizzato, ne esistono di vari modelli con diverse forme per poter eseguire le pieghe con gradi diversi e per evitare di avere collisioni quando si effettua la operazione di piega. Il foglio di lamiera nel corso della relazione sarà denominato anche **elaborato** e semplicemente **pezzo**. In figura 2.2 è mostrata una rappresentazione

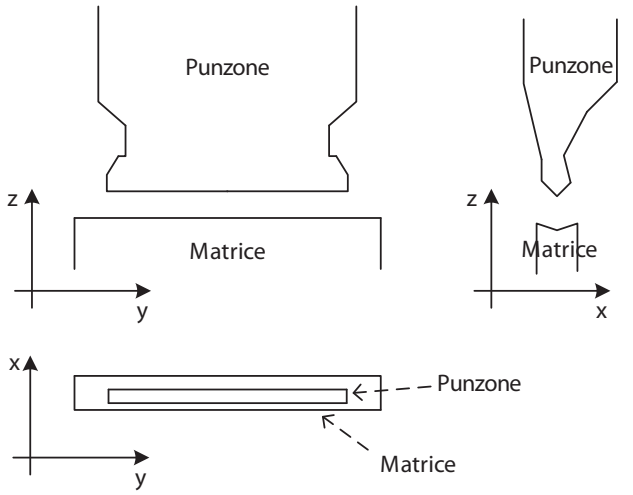

Figura 2.2: Rappresentazione schematica della pressa

schematica di come sarà rappresentata la pressa nei futuri capitoli e come saranno rappresentati gli assi.

2.3 Operazioni del processo di piega

Il processo di piegatura è composto dalle seguenti fasi:

1. **caricamento del pezzo**: un agente esterno alla cella inserisce il pezzo all'interno del caricatore. Il robot controlla la presenza del pezzo e quindi prende la lamiera dal caricatore;

2. **allineamento del pezzo**: il manipolatore posiziona il pezzo all'interno dell'allineatore per poi riprenderlo. Questa operazione è eseguita per determinare la posizione del pezzo nel modo più corretto possibile;

3. **esecuzione della piega**: queste sono le fasi che sono eseguite ogni volta che si esegue una operazione di piegatura:

   (a) **seleziona attrezzaggio macchina**: sono selezionati gli strumenti da utilizzare per effettuare l'operazione di piegatura (matrice e punzone);

   (b) **ribaltamento**: nel caso il robot abbia preso il pezzo dal lato sbagliato, rispetto al tipo di piega successiva, il robot posiziona il pezzo nel ribaltatore per poi prenderlo nel verso giusto;

   (c) **posizionamento in pressa**: il robot posiziona il pezzo nella pressa;

   (d) **esecuzione piega**: la pressa scende ed è eseguita la vera e propria operazione di piega;

   (e) **ripresa pezzo**: il robot riprende il pezzo, nel caso lo avesse lasciato nella operazione precedente e quindi riporta il pezzo in una situazione di sicurezza al di fuori del campo di azione della pressa;

4. **pallettizzazione**: il pezzo completo deve essere spostato nel pallet, la configurazione del pallet è definita dall'utente.

Il simulatore terminato il calcolo delle operazioni esegue una fase di post processing, nella quali i risultati ottenuti sono trasformati in un programma robot che può essere direttamente eseguito dal robot e può essere modificato manualmente.

2.4 Rappresentazione degli oggetti tridimensionali

Tutti gli oggetti presenti nella cella gestiti nel simulatore sono rappresentati attraverso catene cinematiche. L'elaborato è quindi rappresentato come una catena cinematica dove ogni nodo rappresenta una faccia. Ad ogni link della

catena cinematica è associato una posizione ed una mesh utilizzata sia per raffigurare l'oggetto nell'ambiente 3D ed è sia utilizzata dal collision detection per capire se è presente una collisione tra due oggetti. Nell'ambiente in cui è stato sviluppato il software è possibile conoscere direttamente solamente le coordinate della bounding box degli oggetti. Informazioni più dettagliate sulle possibili forme del punzone o della matrice potrebbero essere calcolate leggendo tutti i punti della mesh, rendendo di fatto queste informazioni non recuperabili. Questo fatto ha limitato di non poco i possibili approcci per risolvere il problema non potendo conoscere l'effettiva forma dell'elaborato e del punzone. Quest'ultimo può avere forme particolari, presentendo diverse sporgenze, in questi casi considerarlo sempre come un rettangolo sarebbe un'assunzione troppo restrittiva .

2.5 Descrizione del problema

Lo scopo di questa tesi è la creazione di un algoritmo di planning per pianificare automaticamente l'operazione di ripresa del pezzo durante l'operazione d'esecuzione della piega. L'algoritmo inizia quando il robot ha già preso la lamiera e la pressa ha già raggiunto la sua posizione finale. L'algoritmo deve quindi trovare una traiettoria priva di collisioni che porti l'elaborato dallo stato iniziale a una posizione di sicurezza, dove non ci possono essere collisioni con la pressa. La soluzione prodotta deve essere una sequenza di configurazioni rappresentati le posizione del robot (essendo le catene cinematiche dell'elaborato e del robot collegate, il movimento del robot determina anche il movimento automatico del pezzo). Il programma in una fase successiva alla pianificazione collega i diversi punti con dei movimenti lineari per generare la traiettoria vera e propria.

Durante formulazione della traiettoria non sono tenuti in considerazione i vincoli rispetto la velocità o accelerazione dei giunti. L'unico vincolo che deve essere rispettato è che tutte le posizioni siano raggiungibili e che non presentino collisioni.

2.6 Operazioni di movimento

Il simulatore prevede due tipi di operazioni che possono essere utilizzate per muovere robot nell'ambiente virtuale.

Il primo metodo consiste nell' impostare il valore della posizione dei giunti del robot, Il secondo metodo, invece, consiste nel muovere l'end effector del robot nello spazio cartesiano e successivamente tramite cinematica inversa calcolare la posizione dei giunti del robot. Quest'operazione è più onerosa in termini di tempo rispetto al primo tipo di comando, ma permette agevolmente di capire dove muovere il robot. Inoltre, una posizione dell'end effector nello spazio cartesiano può corrispondere a più posizioni nello spazio dei giunti. Utilizzando questo fatto un algoritmo di planning, in certe condizioni, potrebbe risultare più veloce considerando gli spostamenti calcolati nello spazio cartesiano.

Nella realizzazione del software sono stati utilizzati principalmente i movimenti calcolati tramite cinematica inversa, mentre sono stati utilizzati i movimenti nello spazio dei giunti per generare posizioni causali.

2.7 Specificità del problema

Il problema presentato è una classica situazione risolvibile tramite degli algoritmi di motion planning. Il problema presenta, comunque alcune peculiarità che rendono i comuni algoritmi inefficienti ed è quindi stato necessario sviluppare un algoritmo apposito. Le maggiori peculiarità sono:

- nella posizione iniziale dell'algoritmo il programma può essere fortemente vincolato, quindi i primi movimenti devono essere particolarmente precisi. Questo fatto da un lato rende più difficile trovare la traiettoria del prodotto, ma da un altro lato la soluzione trovata è la soluzione ottima;

- le facce dell'elaborato sono sempre rettilinee, questo fatto garantisce che nella stragrande maggioranza dei casi è necessario eseguire solamente dei movimenti rettilinei per estrarre il pezzo;
- la maggior parte dei problemi sono banali, questo fatto rende inutile utilizzare algoritmi complessi come l'ARW che troverebbero la soluzione, ma impiegandoci una quantità di tempo considerevole; Sarebbe necessario quindi sviluppare un algoritmo che analizza quanto è complesso il problema ed esegue l'algoritmo di planning più opportuno;
- Distanze di sicurezza: la traiettoria deve garantire che l'elaborato non strisci contro il punzone o contro la matrice e rispetti, se possibile, una distanza di sicurezza dagli altri elementi

2.8 Algoritmi di planning presentati in SimulEasy2

Nel simulatore SimulEasy2 sono presenti diversi algoritmi di planning:

- **algoritmo ARW**, si veda il capitolo successivo per una descrizione dettagliata.
- **algoritmo Naive**, tramite il quale si tenta direttamente di connettere lo stato di partenza con lo stato finale. Nel caso di collisione si prova ad eseguire un aggiustamento della traiettoria dal punto di collisione, eseguendo una serie di movimenti perpendicolari alla traiettoria. Questo algoritmo è estremamente veloce, ma funziona solamente quando la traiettoria da trovare è particolarmente semplice;
- **algoritmo bisection**, nel quale sono generati dei punti casuali e si prova a connettere la posizione di partenza con la posizione di arrivo. Se il punto può essere connesso con entrambi allora l'algoritmo termina, se può essere connesso solo con uno si riesegue l'algoritmo per la sezione in cui non è stato possibile trovare la nuova posizione. L'algoritmo esegue una ricorsione fino a trovare un cammino valido o termina quando trova

un punto che non può essere connesso ne con la posizione di partenza ne con la posizione d'arrivo.

Al termine dell'esecuzione di ogni algoritmo è eseguita una fase di smoothing nella quale i punti della traiettoria trovata sono equi distanziati. Questa fase è necessaria per favorire la creazione della vera e propria traiettoria connettendo i punti trovati con i movimenti rettilinei.

Capitolo 3

Algoritmi di pianificazione

In questo capitolo sono descritti gli algoritmi di pianificazione del movimento comunemente utilizzati in robotica autonoma. Questa analisi è necessaria per far capire perché questi algoritmi non sono stati scelti nello sviluppo dell'algoritmo e quali idee, invece, sono state considerate ed utilizzate. Questa breve sintesi non è esaustiva sono considerati solamente gli algoritmi più importanti e non sono considerate le diverse ottimizzazioni o le varianti dei maggiori algoritmi. In questa trattazione non sono descritti algoritmi basati su campi potenziali o algoritmi basati su meta-euristiche come potrebbero essere gli algoritmi genetici. Nello sviluppo del software sono stati impiegati algoritmi di pianificazioni basati sul campionamento dello spazio di configurazione tramite metodi probabilistici.

3.1 Definizione del problema

Il problema di pianificazione del movimento nella sua formula più generale possibile può essere considerato formato da 6 elementi:

- **Spazio di configurazione** uno spazio topologico X,
- **Valori limite** $x_{init} \in X$ e $X_{goal} \subset X$,

- **Collision detector** una funzione $D : X \rightarrow \{true, false\}$, che determina se i vincoli globali sono rispetti dal punto x; Questa funzione può essere a valori binari o una funzione che restuisce valori reali;

- **Input** un insieme, U, che spefica l'insieme dei controlli o azioni che possono modifcare lo stato;

- **simulatore incrementale** dato uno stato corrente $x(t')$, e gli input in un determinato intervallo di tempo $u(t')|t \leq t' \leq t + \Delta t$, calcola $x(t + \Delta t$;

- **una metrica** una funzione $\sigma : X \times X \rightarrow [0, \infty)$.

Il problema di pianificazione del movimento consiste nel trovare un cammino C, che parte da x_{init} e arriva in X_{goal}, in cui ogni punto rispetta i vincoli imposti dal *collision detection*. Nel caso di motion planning per un robot manipolatore lo spazio degli stati è composto dall'insieme delle posizioni dei giunti. I *collision detection* controlla che l'oggetto spostato nel nuovo stato non sia in collisione con gli altri oggetti. Il problema di pianificazione come è stato definito in questa sezione appartiene alla classe dei problemi PSPACE-hard [4]. Per questo motivo per risolvere il problema è necessario l'utilizzo di numerose euristiche ed approssimazioni.

3.2 Pianificazione in spazi continui

Essendo lo spazio di configurazione, nel quale può muoversi il robot, infinito e non numerabile, un algoritmo di pianificazione del movimento dovrebbe provare tutte le possibili configurazioni per trovare la soluzione ottima. Per soluzione ottima si intende che il cammino oltre ad essere fattibile, ovvero che rispetti tutti i vincoli imposti dal *collision detection*, sia il cammino di lunghezza minima, calcolata utilizzando la metrica specificata nel problema. Quando si sviluppa un algoritmo di pianificazione del movimento si richieda che venga trovata una posizione in un tempo ragionevole, in pratica si

richieda che l'algoritmo di pianificazione trovi la soluzione nel minor tempo possibile. In robotica autonoma la soluzione deve essere trovata in tempo reale per garantire un funzionamento corretto del robot. Per quanto riguarda il simulatore di celle industriali, invece, non è richiesto un vincolo così stretto, ma si richiede comunque che l'algoritmo sia veloce, per rendere più usabile il software.

Per rispettare queste richieste gli algoritmi di pianificazione devono necessariamente provare un numero limitato di posizioni. Lo spazio di configurazione deve essere, quindi, campionato. Una possibile tecnica consiste nel dividere lo spazio di configurazioni tramite un algoritmo di campionatura, successivamente utilizzando un algoritmo pianificazione per spazi di configurazione discreti viene trovata una soluzione. Questo approccio ha il grave inconveniente che è necessario campionare tutto lo spazio ed inoltre, se lo spazio non è campionato in modo adeguato la soluzione trovata potrebbe essere molto peggiore rispetto alla soluzione ottimale. Per evitare questi problemi sono stati creati alcuni algoritmi di pianificazione che eseguono il campionamento dello spazio in modo casuale e controllano man mano se sia presente una soluzione.

Nelle successive due sezioni sono descritti due diversi approcci: l'algoritmo ARW e l'algoritmo di pianificazione basato su alberi RRT. Il primo metodo è descritto, perché già presente nelle librerie di SimulEasy2 ed è utilizzato per pianificare dei movimenti da parte del robot manipolatore, ma non trova una cammino fattibile per estrarre l'elaborato dalla pressa quando il pezzo è in una posizione particolarmente difficile. L'idea del secondo algoritmo è, invece, alla base dell'algoritmo realizzato nel corso della tesi.

3.3 Algoritmo di pianificazione ARW

L'algoritmo di pianificazione ARW (Adaptive Random Walk) è stati proposto da Carpi e Pillonello nel 2005 [1]. L'algoritmo crea una cammino in modo casuale partendo dalla posizione di partenza. Ad ogni iterazione è ge-

nerata una posizione casuale, utilizzando una distribuzione gaussiana, vicino all'ultima posizione del cammino. Nel caso il segmento che connette l'ultima posizione del cammino con il punto calcolato non collida con nessun ostacolo, il nuovo punto è inserito nel cammino nell'ultima posizione. Sono generati nuovi punti fino a quando si raggiunge X_{goal}. ll cammino risultante deve essere quindi semplificato, potendoci essere punti inutili che possono essere rimossi, nella pubblicazione si utilizza il termine eseguire una operazione di *smoothing*. L'operazione di smoothing è eseguita utilizzando un algoritmo ricorsivo che prova a connettere due punti del cammino, se è possibile connetterli senza trovare una collisione, allora tutti i punti intermedi sono eliminati dal cammino. Nel caso, invece, sia presente una collisione, l'algoritmo di smoothing è rieseguito due volte prendendo come punti il primo punto e quello intermedio, la prima volta, e il punto intermedio e il secondo punto, la seconda volta. L'algoritmo per adattarsi allo spazio di configurazione in cui opera esegue degli aggiustamenti al metodo utilizzato per generare i punti casuali. La matrice di covarianza è modificata in base ai valori precedenti calcolati per i quali non è stata riscontrata una collisione. Per questo motivo l'algoritmo è chiamata *Adaptive random walk*.

Quest'algoritmo è presente nelle librerie di SimulEasy2 e in molti casi funziona bene. Per quanto riguarda, invece, la pianificazione dell'operazione di estrazione del pezzo ha dato risultati insoddisfacenti, non riuscendo a trovare una soluzione nei casi peggiori. L'algoritmo, inoltre, ha il difetto che nei casi in cui l'estrazione è banale e richiede solamente alcuni movimenti l'algoritmo è lento, essendo necessario generare un cammino composto da molti punti per individuare un cammino vicino a quello ottimale.

3.4 Algoritmo di planning basato su RRT

L'algoritmo proposto è una variante dell'algoritmo di planning basato su RRT (*Rapid exploring Random Tree*) di LaValle [2]. In questa sezione è descritto brevemente l'algoritmo, non sono descritte le varianti o le possi-

Algoritmo 1: ARW

input : x_start

output: Σ il cammino tra x_{start} e X_{goal}

$k \longleftarrow 0$;

$x_k \longleftarrow x_{start}$;

$\Sigma_0 \longleftarrow \Sigma_{init}$;

while *NOT* x_k *in* X_goal **do**

  genera un nuova posizione $s \longleftarrow x_k + random()$;

  if *il segmento che connette* x_k *e* s *giace interamente in* C_{free}

  then

    $k \longleftarrow k + 1$;

    $x_k \longleftarrow s$;

    $\Sigma_k \longleftarrow Update(x_k, x_{k-1}, x_{k-2}, ..., x_{k-H})$;

  end

  else

    elimina S;

  end

end

Algoritmo 2: Funzione di smooth

input : V vettore dei punti da ottimizzare, a, b, S

output: S la lista dei punti ottimizzata

if $a = b$ **then**
  $S.pushBack(V[a])$;
end
else if $a = b$ - *1* **then**
  $S.pushBack(V[a])$;
  $S.pushBack(V[b])$;
end
else if *il segmento* $V[a]$ *e* $V[b]$ *giace interamente in* C_{free} **then**
  $S.pushBack(V[a])$;
  $S.pushBack(V[b])$;
end
else
  $SMOOTH(V, a, (a+b)/2, S)$;
  $SMOOTH(V, (a+b)/2, b, S)$;
end

bili ottimizzazioni dell'algoritmo non essendo state utilizzate nello sviluppo del programma. L'idea principale dell'algoritmo è di creare un albero i cui nodi rappresentano delle posizioni nello spazio del robot. Le posizioni del robot sono generate casualmente e sono inserite nell'albero solamente se è possibile collegare il nodo a un altro nodo dell'albero senza entrare in collisione con gli altri oggetti. Al termine della costruzione dell'albero si cerca di connettere la posizione di arrivo con un nodo dell'albero RRT, se questa operazione è fattibile allora ripercorrendo l'albero al contrario dalla posizione d'arrivo è possibile ottenere un cammino che va dalla posizione di partenza alla posizione di arrivo.

Nei listati 3 e 4 è riportato il codice delle due funzioni necessarie per creare l'albero RRT.

Algoritmo 3: BuildRRT

input : x_{init}
output: T l'albero RRT

$T.init(x_{init})$;
for $k = 1$ *to* K **do**
  $x_{rand} \longleftarrow RandomState()$;
  $Extend(T, x_{rand})$;
end
return T ;

L'algoritmo attraverso la funzione `BuildRRT` costruisce l'albero tentando di generare k nodi. La funzione `Extend` riceve in input l'albero e un nodo generato in modo casuale. `Nearestneighbor` trova il nodo dell'albero posto più vicino al nodo x_{rand}. A questo punto si prova a connettere la posizione generata casualmente con il nodo dell'albero che è più vicino. Questo passaggio è eseguito dalla funzione `NewState` che restituisce il nodo x_{new} la posizione più lontana da x_{near} raggiungibile percorrendo il segmento da x_{near} a x_{rand} prima di avere una collisione. Il nuovo nodo trovato è inserito, quindi, all'interno dell'albero ed è aggiunto l'arco che va da x_{near} a x_{new}. Nel caso

Algoritmo 4: Espandi nodo

input : T, x_{rand}

output: T l'albero RRT aggiornato, una flag se l'operazione è avvenuta

$x_{near} \longleftarrow NearestNeighbor(x_{rand}, T)$;
if $NewState(x, x_{near}, u_{near}$ **then**
    $T.addVertex(x_{new})$;
    $T.addEdge(x_{near}, x_{new}, u_{new})$;
    if $x_{new} = x$ **then**
        return *raggiunto* ;
    end
    else
        return *avanzato* ;
    end
end
return *bloccato* ;

non si volesse eseguire tutte le iterazioni, l'algoritmo potrebbe essere terminato quando è possibile connettere direttamente l'ultimo nodo a un punto in X_{goal}. Nella maggior parte dei casi è preferibile creare due alberi, uno che ha come radice x_{start} e un altro che ha come radice x_{end}. L'algoritmo genera in modo alternato un nodo per ogni albero. Quando è creato un nodo si cerca di connetterlo con il nodo più vicino dell'altro albero. Se questa operazione è fattibili è stato trovato un cammino fattibile l'algoritmo può terminare.

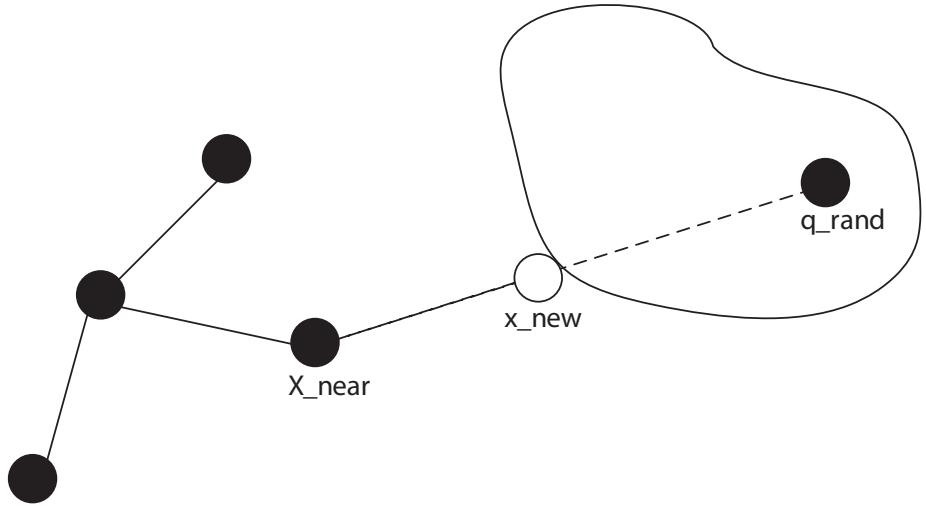

Figura 3.1: Creazione di un nuovo nodo in albero RRT

Capitolo 4

Realizzazione del software

4.1 Introduzione

In questo capitolo è spiegato l'algoritmo realizzato. Nella prima parte del capitolo sono descritti i tentativi effettuati per realizzare il software e i risultati ottenuti. Nella seconda parte del capitolo è descritto in modo dettagliato il software realizzato.

4.2 Sviluppo dell'algoritmo

Durante lo sviluppo del software soprattutto nelle prime fasi, sono stati provati diversi approcci per trovare la soluzione che meglio si adattasse a questo particolare tipo di problema. Nell sezioni sottostanti sono descritti gli approcci utilizzati e i risultati ottenuti. Questi tentativi sono descritti per due motivi: per spiegare in dettaglio come è stato realizzato il lavoro ed elencare le difficoltà trovate, ma soprattutto perché molte idee degli algoritmi sono state utilizzate nella scrittura del software finale.

4.2.1 Primo approccio: utilizzo dell'algoritmo ARW

Nelle librerie del software SimulEasy2 sono presenti alcuni algoritmi di pianicazione, tra di questi quello più complesso, che è stato scritto appositamente

per risolvere i casi più difficili è l'ARW (nella sezione 3.3 è descritto in dettaglio il suo funzionamento). Prima di sviluppare il software e di analizzare il problema si è adoperato questo algoritmo, per controllare se potesse essere utilizzato o fossero solamente necessarie alcune modifiche.

4.2.2 Secondo approccio: realizzazione algoritmo basato su RRT

L'algoritmo ARW ha il problema di non provare a generare i punti in modo casuale e provarli a connettere direttamente ad un cammino. Questo algoritmo ha il difetto di funzionare male nelle situazioni in cui bisogna imboccare e percorrere un lungo cammino, come descritto in [2]. Nella posizione inziale in cui il pezzo è fortemente vincolato la soluzione e equiparabile a imboccare un lungo cammino per uscire da una stanza. Per questo motivo si è deciso di provare ad utilizzare un algoritmo di pianificazione basato su RRT. La speranza di questo approccio era che utilizzando un albero per memorizzare tutti i tentativi eseguiti fosse più facilmente rilevabile il cammino lungo il quale uscire. L'algoritmo RRT è stato scritto da zero, utilizzando la libreria Flann (*Fast Library for Approximate Nearest Neighbors*), per memorizzare quali aree dello spazio di configurazioni fossero già state visitate e per trovare il nodo dell'albero più vicino allo stato casuale generato. Si è deciso di utilizzare come spazio di configurazione i valori dei giunti del robot, per evitare di dover eseguire ad ogni movimento un valore di cinematica inversa. I risultati dell'algoritmo sono stati abbastanza deludenti. Essendo l'elaborato fortemente vincolato nella posizione iniziale è necessario all'inizio generare dei movimenti piccolissimi. Questo comporta nel dover generare un numero altissimo di punti (oltre i 10000) per vedere un reale miglioramento, nel quale il pezzo è stato parzialmente estratto.

Per evitare questi problemi si è provato a realizzare un algoritmo nel quale non fossero generati dei nuovi punti in modo casuale, ma piuttosto si tentasse di muovere un giunto del robot fino a rilevare una collisione e quindi selezionare il punto prima della collisione. La posizione di partenza è sele-

zionata in modo casuale. Utilizzando questo metodo è stato possibile vedere un significativo miglioramento riuscendo effettivamente a liberare il pezzo in circa 20 iterazioni. Il problema di questo approccio è che non tenendo conto delle posizioni intermedie, i giunti sono mossi più del necessario ed, inoltre, cosa ancora più importante c'è la possibilità di raggiungere posizioni di stallo. L'ultimo tentativo è stato condotto un approccio ibrido generando sia posizioni casuali che provando a muovere i giunti fino al loro limite massimo. Questo approccio non ha portato a dei miglioramenti significativi, per la difficoltà di a impostare i valori per creare l'albero e quali nodi selezionare dell'albero per muovere i giunti fino al loro limite. Quest'approccio, inoltre, ha il grave difetto di essere dipendente dalla tipologia del problema, qualora si cambiasse l'elaborato i parametri trovati potrebbero non essere più ottimali e l'algoritmo dovrebbe essere quindi riconfigurato, rendendo di fatto l'algoritmo del tutto inutilizzabile.

4.2.3 Terzo approccio: Algoritmo di pianificazione basato su RRT con parametri adattivi

Per risolvere i problemi riscontrati nello sviluppo dell'algoritmo di planning basato su RRT. Si è deciso di provare a sviluppare alcune euristiche per selezionare i nodi dell'albero nei quali provare ad eseguire le operazione di movimento dei giunti fino al raggiungimento delle collisioni. Oltre a questo si è tentato anche di adattare i parametri per la generazione dei punti casuali in base al numero di nodi che si è tentato di costruire e in base alla distanza dell'ultima posizione trovata rispetto al punzone e rispetto alla matrice.

In figura 4.1 è rappresentato uno schema raffigurante le operazione compiute dal programma. Per risolvere i problemi riscontrati nello sviluppo dell'algoritmo di planning basato su RRT. Si è deciso di provare a sviluppare alcune euristiche per selezionare i nodi dell'albero nei quali provare ad eseguire le operazione di movimento dei giunti fino al raggiungimento delle collisioni. Oltre a questo si è tentato anche di adattare i parametri per la generazione dei punti casuali in base al numero di nodi che si è tentato di costruire e in

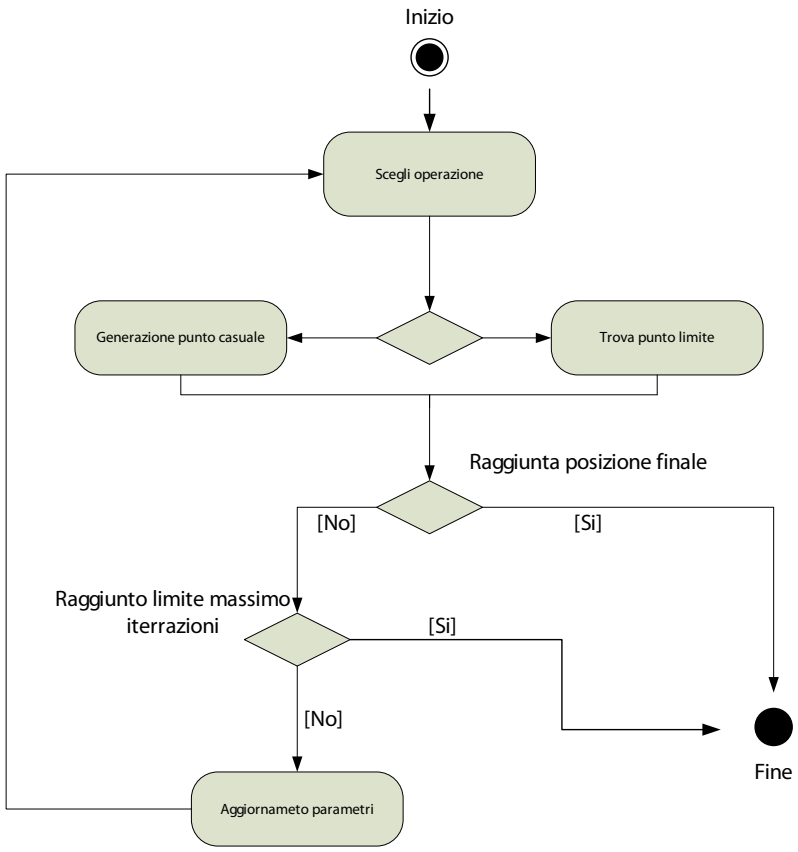

Figura 4.1: Activity diagram algoritmo tentativo 3

base alla distanza dell'ultima posizione trovata rispetto al punzone e rispetto alla matrice. In figura è rappresentato uno schema raffigurante le operazione compiute dal programma. La principale difficoltà consiste nella configurazione dei parametri, che determinano fortemente l'efficacia dell'algoritmo. I risultati ottenuti, inoltre, sono stati leggermente migliori rispetto agli altri casi, ma ancora del tutto insoddisfacenti, è stato quindi necessario provare altre strade.

4.2.4 Quarto approccio: algoritmo finale

Negli approcci precedenti non si è mai considerato che l'algoritmo di pianificazione il più delle volte deve risolvere delle situazioni banali, in cui sono solamente necessarie dei semplici movimenti rettilinei. Per cercare di rendere l'algoritmo efficiente anche nelle situazioni più semplici e che non necessiti una fase di configurazione, si è deciso di utilizzare un approccio completamente diverso rispetto agli altri tentativi. Sono state codificate una serie di operazioni elementari tramite le quali tentare di spostare l'elaborato per raggiungere una nuova posizione. Utilizzando queste operazioni ed eseguendo diversi tentativi è stato possibile realizzare un programma che permettesse di selezionare in modo veloce una sequenza di operazioni per estrarre il pezzo. Il problema di codificare delle operazioni precise è che l'algoritmo perde del tutto la generalità richiesta e in alcune situazione potrebbe non generare alcuna soluzione. Per ovviare a questo problema è stato necessario studiare con attenzione le possibili operazioni e trovare una sequenza quanto più generale possibile. Oltre a questo ogni nuova posizione trovata è memorizzata all'interno di un albero e qualora le operazioni non dessero il risultato sperato, sono generate delle posizioni casuali, rendendo di fatto l'algoritmo identico a un classico algoritmo RRT. La struttura dell'algoritmo è identica a quella realizzata nel terzo approccio, dove al posto di realizzare dei movimenti in base al valore massimo raggiungibile muovendo un giunto è selezionata un'operazione complessa predeterminata. Le operazioni codificate sono molto efficaci, quindi sono generate delle posizioni casuali solamente se le operazio-

ni non riescono a trovare una nuova posizione. Una situazione abbastanza rara. Nelle successive sezioni è descritto l'algoritmo più in dettaglio.

4.3 Struttura dell'algoritmo

L'algoritmo è realizzato attraverso quattro funzioni:

1. **ChoosePosition** seleziona quale operazione eseguire, aggiorna l'albero e decide se provare ad estrarre direttamente il pezzo con `extractSimple`;
2. **ExeOperation** cerca di trovare una nuovo stato che non è stato precedentemente già visitato partendo dalla posizione scelta con `ChoosePosition`;
3. **ExtractSimple** prova a estrarre l'elaborato fino alla posizione finale;
4. **AdjustMetrics** corregge i parametri utilizzati nelle funzioni precedenti.

In figura 4.2 è mostrato l'*activity diagram* del programma. Il programma termina se è trovata una traiettoria fattibile o se il numero d'iterazioni ha superato il limite imposto, in questo caso non viene restituita alcuna traiettoria, anche se potrebbe essere stata trovata una traiettoria parziale. Nel diagramma con il termine operazioni non riuscita si intende che non è stata trovata una nuova posizione o che è stata trovata ma è troppo vicina alla posizione di partenza.

4.4 ChoosePosition

La funzione riceve come input lo stato del mondo ottenuto dall'ultima operazione eseguita, e, quindi, è eseguito un confronto tra l'ultima posizione ottenuta con le altre calcolate in precedenza e memorizzate nell'albero. Il confronto avviene in base a quattro valori:

1. **metrica uno** la differenza rispetto alla posizione precedente, calcolata facendo la differenza del valore dei giunti;

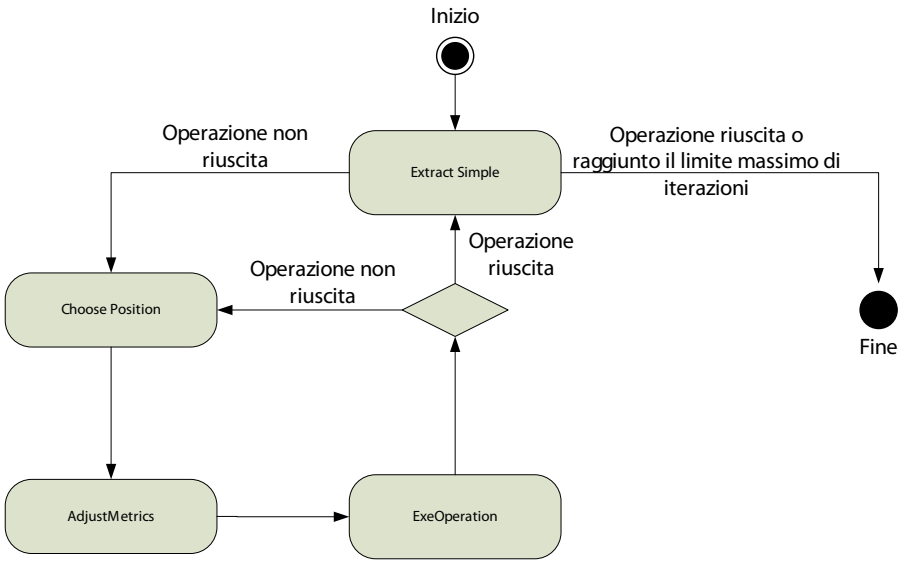

Figura 4.2: Activity diagram dell'algoritmo di estrazione

2. **metrica due** la distanza del punto trovato rispetto al punto precedentemente trovato che è più simile, questo calcolo è eseguito utilizzando un *kdtree*;

3. **metrica tre** è la distanza del punto trovato rispetto al punzone in questo modo è garantito che si sceglie sempre il punto più lontano dalla pressa;

4. **metrica quattro** la distanza del punto trovato rispetto alla matrice, garantisce che il pezzo non vada a scontrarsi contro la matrice.

I primi due valori sono utilizzati per capire se il nuovo valore calcolato determina davvero una nuova posizione, la metrica numero uno è utilizzata per capire quale dalle nuove posizione calcolate modifica maggiormente la situazione. La metrica numero due è utilizzata per calcolare che non siano trovate più volte configurazioni simili. Le metriche tre e quattro sono utilizzate quindi per calcolare quale posizione distanzia maggiormente l'elaborato dalla pressa. Nel listato 5 è rappresentato il codice di come le metriche sono utilizzate.

Algoritmo 5: ChoosePosition

input : la nuova posizione trovata

output: la posizione scelta

if *distance(newPosition, previousPosition)* $\leq$ *limitDistance* **then**

    Scegli newPosition ;

end

else if *newPosition diversa dalle posizioni precedentemente trovate* **then**

    seleziona la posizione che ha la distanza maggiore tra la matrice e il punzone ;

end

else

    imposta previousPosition come visitata ;

    seleziona una soluzione precdentemente calcolata che ha la distanza maggiore tra la matrice e il punzone ;

end

Queste sono le operazioni eseguite per determinare la nuova posizione, in pratica, essendo la funzione `ExeOperation` particolarmente efficace è quasi sempre scelta l'ultima posizione generata. Questo fatto è una garanzia della bontà della funzione `ExeOperation` ed, inoltre, garantisce che la traiettoria trovata non presenta operazioni inutili ed è quindi molto vicina alla soluzione ottimale (si veda la sezione per un'analisi precisa dei risultati ottenuti 5.1).

4.5 ExtractSimple

La funzione `extractSimple` prova ad eseguire una operazione di estrazione naive cercando semplicemente di distanziare il pezzo dalla matrice e dal punzone e quindi di allontanarlo dalla pressa. L'operazione di distanziamento verticale è eseguita per assicurarsi che il pezzo durante l'operazione di estrazione non strisci con il punzone o con la matrice. Questa operazione è eseguita ad ogni ciclo, solamente se la nuova posizione trovata è completamente diversa. Quando la funzione termina correttamente si è allora trovata una traiettoria di estrazione fattibile. L'algoritmo quindi termina. In figura 4.3 è mostrata l'esecuzione dell'operazione. Le frecce rosse rappresentano la direzione lungo la quale è eseguito il movimento.

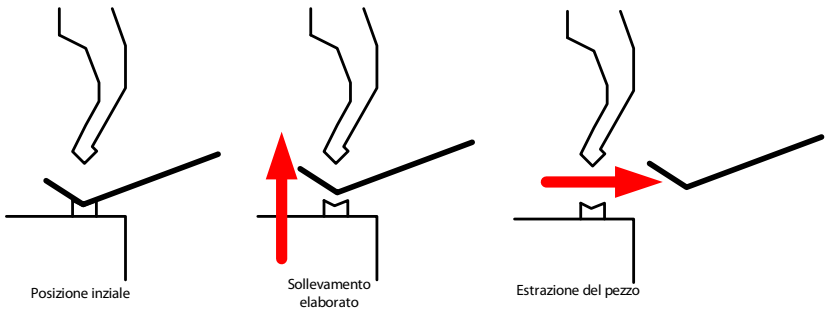

Figura 4.3: Operazioni eseguite dall'operazione extract simple

4.6 ExeOperation

La funzione `exeOperation` è il nucleo centrale del programma. La funzione prende come input una particolare configurazione del problema e restituisce una posizione che migliora almeno uno dei seguenti valori:

- la distanza dell'elaborato dal punzone;
- la distanza dell'elaborato dalla matrice;
- la distanza rispetto alla posizione precedente
- la distanza dalla pressa in generale.

Con il termine migliore si intende che la nuova distanza calcolata è maggiore rispetto alla posizione precedente. Questo non vuol dire che la nuova posizione non necessariamente appartiene a un cammino lungo il quale è possibile estrarre l'elaborato. Questi valori indicano, invece, che è si riusciti a muovere il pezzo e quindi bisogna provare a seguire quella direzione.

Per raggiugere questo scopo sono state codificati cinque tipi di movimenti:

- **Fase 1** tentativo lungo l'asse globale orizzontale:
- **Fase 2** tentativo lungo l'asse globale verticale;
- **Fase 3** tentativo lungo direzioni parallele alla faccia dell'elaborato a cui è collegato l'end effector;
- **Fase 4** rotazione dell'elaborato rispetto al punzone.
- **Fase 5** movimenti casuali.

La funzione prova a muovere l'elaborato attraverso ognuno di questi possibili movimenti. In figura 4.4 è mostrato come sono provati i diversi movimenti. L'idea centrale dell'algoritmo è quella di selezionare una serie di movimenti elementari, che risolvono immediatamente alcuni casi banali, mentre quando si affrontano casi particolarmente complessi sono generati questi movimenti,

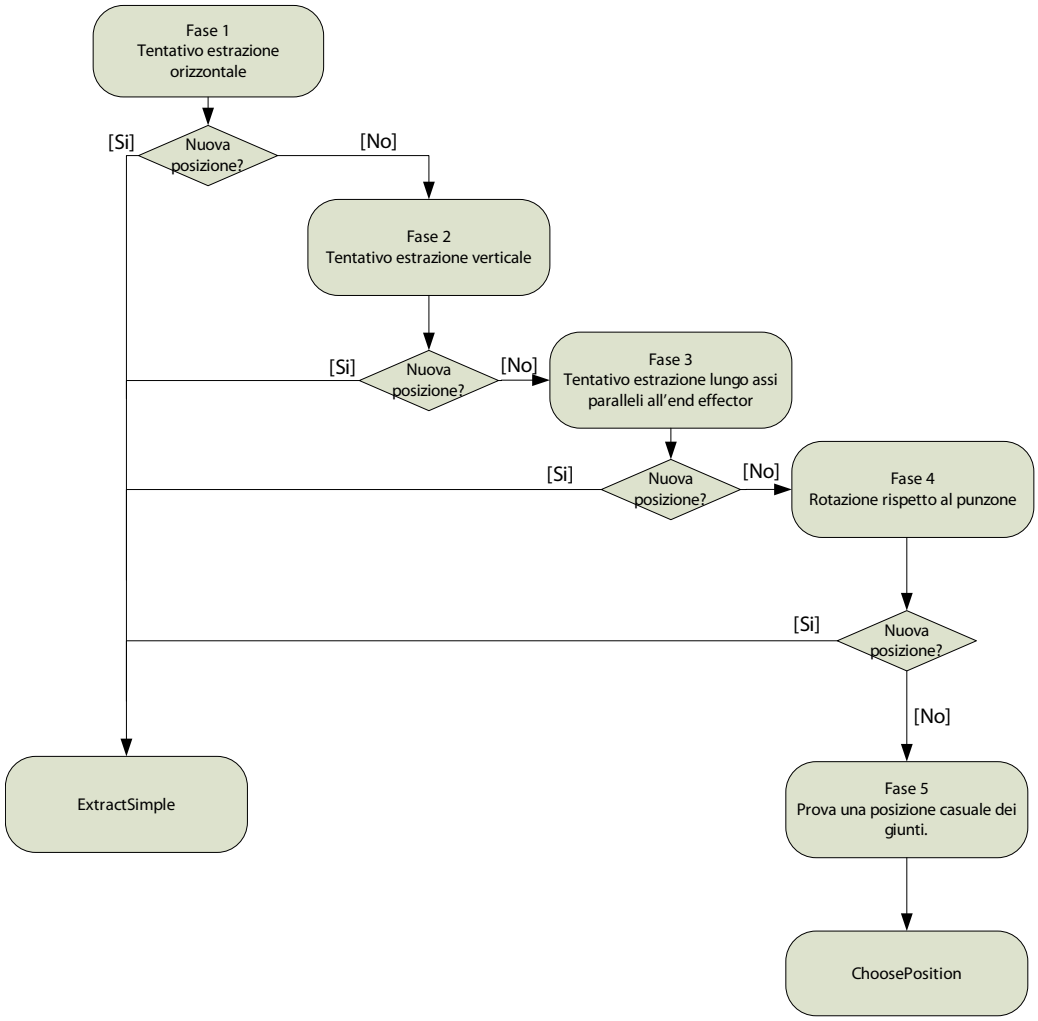

Figura 4.4: Activity diagram funzione ExeOperation

che se possono essere eseguiti, garantiscono che venga trovata una posizione che appartiene al cammino finale con una buona probabilità.

Ogni nuova posizione trovata che è sufficientemente diversa da tutte le posizioni precedenti è memorizzata nel *kdtree*, in modo tale che possa essere adoperata in un ciclo successivo qualora non sia siano state trovate delle nuove migliori posizioni. Con il termine sufficientemente diversa vuol dire che la distanza da una configurazione dei giunti del robot rispetto a alla configurazione più simile è maggiore di un certo valore. Questa operazione è possibile eseguirla velocemente utilizzando la libreria Flann (Fast Library for Approximate Nearest Neighbors [3]). Le distanze dei movimenti sono calcolate in base al numero delle iterazione in cui è eseguito il programma e rispetto e al numero delle nuove posizioni trovate. Questi valori sono calcolati dalla funzione `AdjustMetrics` si veda la sezione 4.6.7 per una descrizione esaustiva. Nelle successive fasi sono spiegate in dettaglio come sono stati realizzati i diversi movimenti e perché sono stati scelti proprio questi movimenti.

4.6.1 Fase 1

In questa fase si cerca di spostare l'elaborato attraverso un movimento parallelo alla faccia della pressa. In figura 4.5 sono mostrate lungo quali direzioni è effettuato il movimento. Questa scelta è data dal fatto che l'elaborato potrebbe avere una faccia laterale il cui movimento impedisce lo spostamento verticale. In figura 4.6 è mostrata una situazione, nella quale il movimento eseguito dalla fase 1 permette di eseguire successiva l'operazione `ExtractSimple` ed estrarre completamente il pezzo, eseguendo un movimento verso destra dell'elaborato

4.6.2 Fase 2

Nella fase 2 il pezzo è spostato lungo l'asse verticale. In figura 4.7 è mostrato il movimento eseguito. Questa fase è eseguita solamente se nella prima iterazione il movimento verticale dell'operazione ExtractSimple è risultato

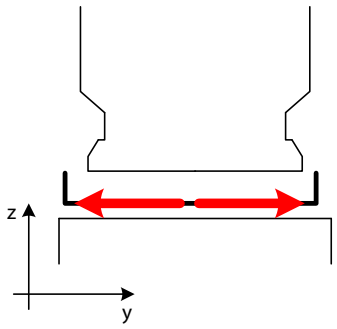

Figura 4.5: Movimenti eseguiti nella fase 1

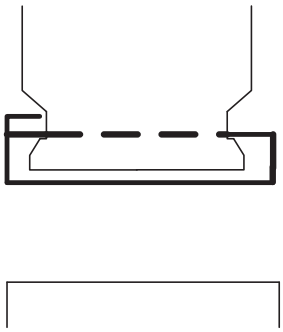

Figura 4.6: Caso in cui è utile il movimento eseguito nella fase 1

fattibile, quindi il punto è raggiungibile dal robot e non sono presenti collisioni. L' operazione è necessaria nel caso ci si trova in una situazione simile a quella mostrata in figura 4.8 nella quale il pezzo deve essere ulteriormente abbassato per poi evitare una collisione, essendoci presente una faccia dietro al punzone. Si ricorda, invece, al lettore che i movimenti lungo l'asse verticale, eseguiti nella operazione ExtractSimple, sono realizzati per evitare che il pezzo strisci contro il punzone o contro la matrice quando si esegue l'operazione di estrazione.

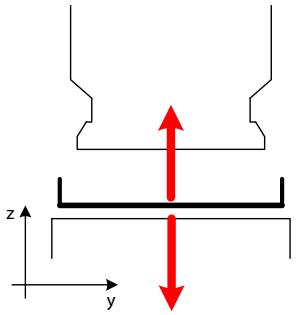

Figura 4.7: Movimenti eseguiti nella fase 2

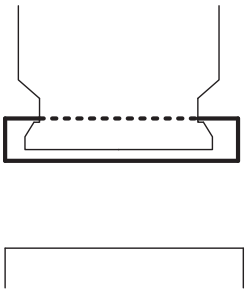

Figura 4.8: Caso in cui è utile il movimento eseguito nella fase 2

4.6.3 Fase 3

In quest'operazione il pezzo è spostato parallelamente rispetto alla faccia del pezzo, lungo gli assi x. Per capire in modo migliore come è eseguita l'operazione si guardi la figura 4.5 nella quale sono mostrate le direzioni lungo

le quali è estratto il pezzo. Il pezzo può essere spostato lungo due direzioni, una verso l'esterno dell'elaborato ed una che porta il pezzo all'interno dell'elaborato. La prima direzione che viene provata è quella che muove il pezzo

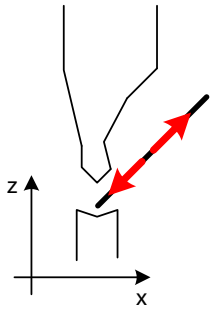

Figura 4.9: Movimenti eseguiti nella fase 3

lontano dalla pressa, questo scelta è effettuata perché se è possibile muovere lungo questa direzione allora successivamente direttamente può essere eseguita l'operazione `extractSimple`, senza eseguire ulteriori passaggi. Lo spostamento che conduce il pezzo all'interno della pressa sono stati impostati dei limiti in modo da garantire che il pezzo non sia spostato troppo all'interno perché oltre un certo limite lo spostamento diventa inutile, rendendo ancora più difficile estrarre il pezzo oltre che essere potenzialmente pericoloso.

La decisione di muovere l'elaborato lungo gli assi è stata scelta perché le facce dell'elaborato sono dritte quindi prive di sporgenze è quindi sempre possibile muovere il pezzo almeno lungo una di queste due direzioni. In figura 4.10 sono mostrati due esempi nel quale questa operazione è l'unica direzione lungo la quale può essere estratto il pezzo, considerando che il pezzo non può essere mosso verticalmente perché sono presenti delle facce che impediscono questo tipo di movimento. Le direzioni possibili sono indicate attraverso una freccia blu.

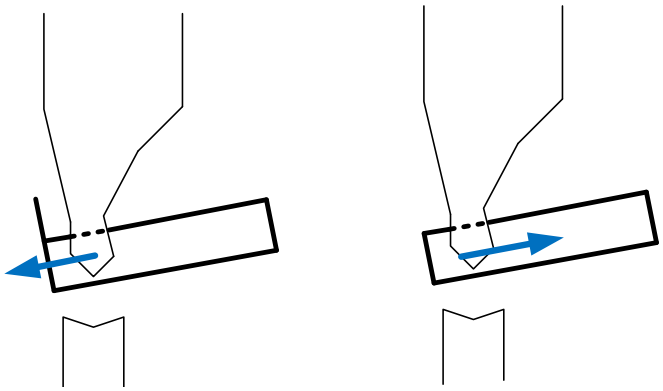

Figura 4.10: Caso in cui è utile il movimento eseguito nella fase 3

4.6.4 Fase 4

Oltre a movimenti rettilinei devono anche essere considerati anche i movimenti di rotazione. In questa fase sono prese in esame solamente le rotazioni dell'elaborato considerando come asse di rotazione lo spigolo inferiore del punzone, in figura 4.11 è mostrato lungo quali parametri avviene la rotazione. Tramite questo movimento si cerca di trovare una nuova posizione qualora non fossero possibili movimenti rettilinei, in figura 4.12 è mostrata una tipica situazione che può essere risolta in modo ottimale attraverso la fase 4. L'angolo di massima per cui si tenta di ruotare il pezzo dalla posizione originale non può superare un certo limite (questo parametro è stato impostato nella fase finale dello sviluppo del software a 45°). Questo valore è stato impostato per evitare che il pezzo raggiunga delle posizioni estreme in cui le facce dell'elaborato sono quasi parallele alle facce della pressa. Queste posizioni permettono di eseguire movimenti molto ampi che però non portano alcun miglioramento, quindi per come è stata pensata la funzione di **ChoosePosition** rischierebbero di rallentare l'algoritmo.

4.6.5 Fase 5

La fase 5 è l'ultimo movimento tentato eseguito solamente se nessuna fase precedente ha dato una soluzione. Durante la funzione è eseguito un movi-

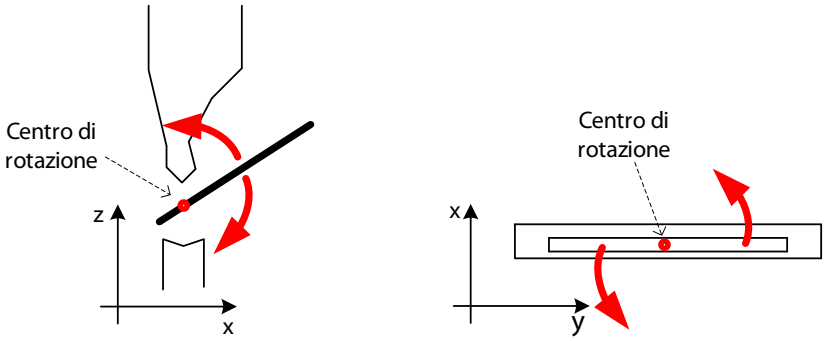

Figura 4.11: Movimenti eseguiti nella fase 4

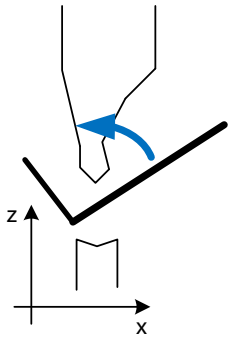

Figura 4.12: Caso in cui è utile il movimento eseguito nella fase 4

mento casuale dei giunti fino a quando si arriva ad una posizione distante oltre una certa soglia dalla posizione inziale. Questa soglia è controllata dalla funzione `adjustMetrics` l'operazione è quindi una extrema ratio difficilmente è adoperata. Si è deciso di sviluppare questa funzione per assicurarsi che il programma possa dare una soluzione anche nelle situazioni particolarmente complesse o che dopo ripetuti movimenti l'elaborato si è bloccato in una posizione nella quale le altre operazioni non hanno dato il risultato sperato.

4.6.6 Movimenti non precisi

I movimenti realizzati nella fase sono stati descritti come movimenti precisi da una posizione iniziale ad una finale. In verità le funzioni sono state sviluppate in modo tale da adattarsi alla situazione qualora non sia trovata una posizione corretta. In pratica quando la posizione in cui si vuole muovere l'elaborato presenta delle collisioni si prova ad eseguire dei piccoli movimenti, lungo direzione diverse da quella in cui si è mosso l'elaborato. Se attraverso questi piccoli movimenti è possibile trovare una configurazione priva di collisioni, la fase termina restituendo la nuova posizione trovata. Per capire meglio questo punto conviene considerare questo esempio. L'elaborato è spostato lungo una direzione parallela alla faccia della pressa. Questa operazione non termina perché si è rilevata una collisione vicino alla posizione finale. Allora si tenta di eseguire dei piccoli movimenti lungo la verticale per vedere se può essere trovata una nuova posizione valida. Ciò potrebbe essere utile nel caso l'elaborato non è perfettamente parallelo alla matrice.

Questi aggiustamenti sono utilizzati soprattutto nelle operazioni di rotazione dell'elaborato. Non conoscendo l'effettiva forma del punzone, durante i movimenti di rotazione è difficile valutare in modo corretto lungo quale ruotare il pezzo. Quindi l'operazione deve essere corretta. Questo fatto è particolarmente fastidioso, perché un'operazione semplice come una rotazione richiede alcuni parametri che devono essere impostati dall'utente. Una loro errata impostazione potrebbe portare a scartare alcuni movimenti validi, determinando un calo delle prestazioni dell'algoritmo o causare che non

riesca a trovare una soluzione, pur essendoci. Le operazioni di movimentazione sono state pensate in modo tale da non avere ridondanze, quindi il movimento dovrebbe essere trovato successivamente utilizzando la fase 5, quindi generando il movimento in modo casuale, ciò rallenta l'algoritmo e in molti casi porterebbe a non trovare più una nuova soluzione.

4.6.7 AdjustMetrics

In questa funzione sono impostati i parametri utilizzati nelle diverse fasi dell'algoritmo in base al numero di iterazione e rispetto al numero delle nuove posizioni trovate. In linea di massima i valori degli spostamenti specificati nelle diverse fasi sono man mano ridotti se non sono trovate nuove configurazione, mentre sono aumentati se sono trovate nuove configurazioni. Oltre alle distanze delle operazioni di movimento varia la distanza minima richiesta tra un nodo del *kdtree* e il nodo che si vuole inserire nel *kdtree*. Anche in questo caso la distanza è ridotta se non sono trovate nuove posizioni valide. Questi aggiustamenti sono realizzati per evitare di rendere l'algoritmo indipendente da questi parametri, in modo tale che non debba essere configurato quando è cambiato il tipo di cella o di robot utilizzato.

Le variazioni dei parametri sono in dettaglio riportate nella tabella 4.6.7. I parametri subiscono una modifica come riportato in tabella 4.6.7 se il valore del parametro non è già il valore massimo e minimo consentito. Questi valori devono essere configurati manualmente, ma non influenzano in modo determinante l'esecuzione del programma. I limiti maggiori non sono molto importanti, perché quando sono trovate un numero cospicuo di nuove posizioni è probabile che l'elaborato sia già in una posizione che garantisce l'utilizzo della funzione `ExtractSimple`. I limiti massimi sono stati quindi impostati per effettuare un ulteriore controllo, nel caso siano sorti dei problemi. I limiti inferiori, sono abbastanza importanti, perché posti troppo grandi possono influenzare la possibilità di trovare una soluzione. Nel caso, invece, che sono scelti troppo piccoli, semplicemente il programma potrebbe eseguire più iterazioni del necessario. Per impostare correttamente i parametri è quin-

di necessario solamente scegliere dei valori molto laschi che non vincolano troppo la generazione delle nuove posizioni.

Parametro	**Trovati nuovi punti**	**Nessun punto trovato**
movimento fase 1	$+2 * a$	$-a$
movimento fase 2	$+2 * b$	$-b$
movimento fase 3	$+2 * b$	$-b$
movimento fase 4	$+2 * b$	$-b$
distanza minima punti differenti	$*2$	$/2$
parametri correzione translazioni	$+2 * c$	$-c$
parametri correzione rotazione	$*2 * d$	$-d$

I valori: a, b, c, d e anche i limiti di ogni movimento sono dei parametri che sono calcolati in modo automatico durante la prima iterazione del programma. Questi valori devono essere molto piccoli, per evitare che variazioni troppo consistenti pregiudichino la scoperta di nuovi punti, quindi è più che sufficiente impostarli come un centesimo degli spostamenti iniziali di ogni movimento.

Per alcuni parametri comunque non è stato possibile trovare una procedura automatica per determinarli, quindi devono essere impostati a mano. Gli unici parametri che devono essere configurati sono i seguenti:

- la distanza allontanamento matrice o dal punzone per cui si muove il l'elaborato nella operazione ExtractSimple;
- la distanza minima per cui bisogna spostare l'elaborato dalla pressa per ritenere conclusa l'operazione di estrazione;
- le distanze degli spostamenti inziali tentati dalle fasi nella prima iterazione del programma.
- parametri di correzione delle rotazioni;

4.7 Completezza

L'algoritmo sviluppato, nel caso siano trovate nuove posizioni solamente attraverso movimenti casuali dei giunti, è un semplice algoritmo di pianificazione basato RRT, quindi la probabilità che sia trovata una soluzione è pari a uno nel caso il numero delle iterazioni tenda a infinito. In pratica l'algoritmo difficilmente non trova una nuova posizione durante le prime quattro fasi. Questo risultato garantisce, inoltre, che le operazioni codificate sono corrette e veramente semplificano il problema e rendono possibile trovare una soluzione in modo più veloce.

Capitolo 5

Risultati

In questo capitolo sono riportati i risultati ottenuti ed inoltre è mostrato un esempio di come funziona l'algoritmo per risolvere una situazione particolarmente complessa.

5.1 Risultati ottenuti

Il programma realizzato ha trovato la soluzione in ogni caso provato, anche in quelli più complessi, e sempre in un tempo ragionevole. Non sono state eseguiti dei test approfonditi, essendo l'algoritmo presentato ancora in fase di prototipizzazione e non ancora completamente integrato al software. Questo è dovuto al fatto che le librerie di pianificazione presenti in SimulEasy2, durante lo sviluppo del software sono state modificate ed aggiunte funzionalità. L'algoritmo dovrà quindi essere ulteriormente sviluppato e poi integrato. Il programma comunque ha dato risultati soddisfacenti nelle varie prove, trovando la soluzione in tempi molto rapidi, inferiori ai 5 secondi. Questo valore è più che sufficiente, essendo l'algoritmo utilizzato da un simulatore. Il programma ha sempre eseguito una poche iterazioni e sono sempre state generate poche posizioni. Questo garantisce che le singole operazioni codificate, ovvero le varie fasi sviluppate, hanno dato i risultati sperati generando solamente posizioni corrette. Il numero di posizioni generate e il numero di iterazioni

sono stati i due parametri che sono tenuti più in considerazione durante lo sviluppo del software, essendo ciò che maggiormente influenza il tempo d'esecuzione del programma. Il programma, si è cercato di svilupparlo in modo tale che non sia necessario configurarlo, essendo tale operazione automatica. Questo è obiettivo, è stato, in certo modo raggiunto, parametri iniziali non determinano se viene o non viene trovata una soluzione, ma determinano il tempo necessario per trovarla. Questo è dovuto al fatto che se i parametri iniziali sono molto diversi dai parametri necessari, è necessario eseguire diverse iterazioni per eseguire `AdjustMetrics` per trovare delle posizioni corrette, nel caso i parametri inseriti siano troppo grandi. Nel caso, invece, i parametri siano troppo piccoli l'algoritmo deve eseguire molte iterazioni, trovando molti punti fattibili. Questo ultimo caso è da evitare quindi è preferibile inserire valori alti e lasciare il programma che poi esegua le dovute correzioni.

Per quanto riguarda le soluzioni prodotte, l'algoritmo non garantisce che il *path* calcolato sia quello ottimale. Il cammini trovati, comunque, non hanno mai presentato movimenti completamente inutili, o raggiunto posizione visibilmente non corrette.

5.2 Esempio

In questa sezione, per meglio illustrare il funzionamento dell'algoritmo, è descritto come l'algoritmo trova la soluzione in una situazione particolarmente complessa. In figura 5.1 è raffigurata la posizione iniziale. Il pezzo ha una forma che impedisce i movimenti verticali perché il punzone presenta delle rientranze, chiamate linguette. In figura 5.2 sono mostrate in dettaglio come sono rappresentate le facce dell'elaborato. Nelle figure successive sono raffigurate le posizioni trovate in quattro iterazioni, che portano all'estrazione completa del pezzo, l'ultima immagine rappresenta il risultato dopo l'esecuzione dell'operazione **ExtractSimple**. La lista delle operazioni eseguite è la seguente:

1. Fase 3, il risultato è mostrato nella figura 5.3;

2. Fase 4, in questa fase si può notare che la posizione trovata è stata determinata tramite una rotazione e una serie di traslazioni per trovare una posizione priva di collisioni. Risultato è mostrato nella figura 5.4;

3. Fase 2, risultato è mostrato nella figura 5.5;

4. `ExctractSimple`, potendo allontanare il prezzo lungo la verticale dal punzone, può quindi essere eseguita l'operazione di estrazione. Risultato è mostrato nella figura 5.6.

L'algoritmo riesce quindi a trovare una soluzione eseguendo solamente 5 iterazioni e non viene generata nessuna posizione casuale.

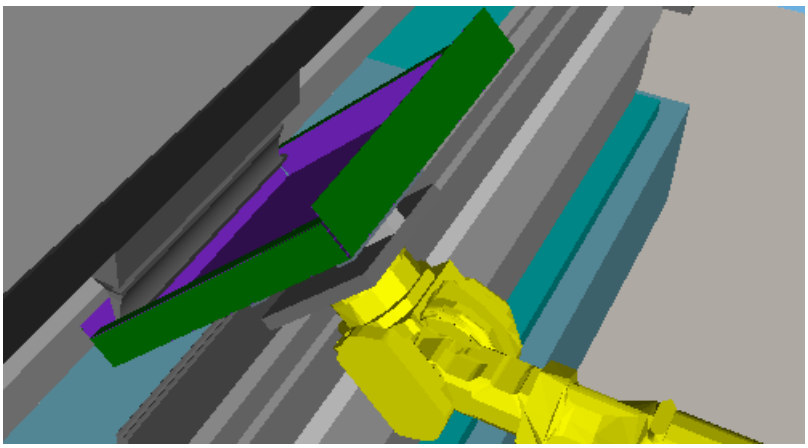

Figura 5.1: Posizione inziale

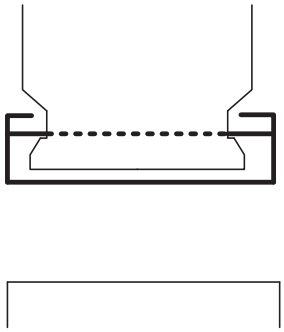

Figura 5.2: Rappresentazione facce laterali

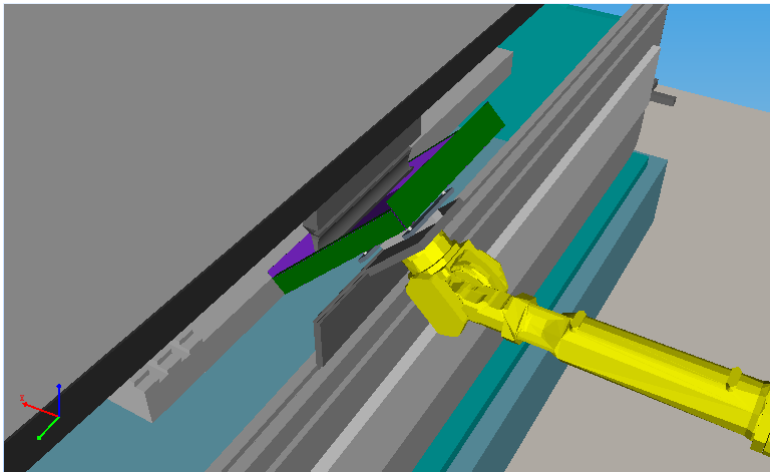

Figura 5.3: Posizione risultate dopo esecuzione operazione **fase 3**

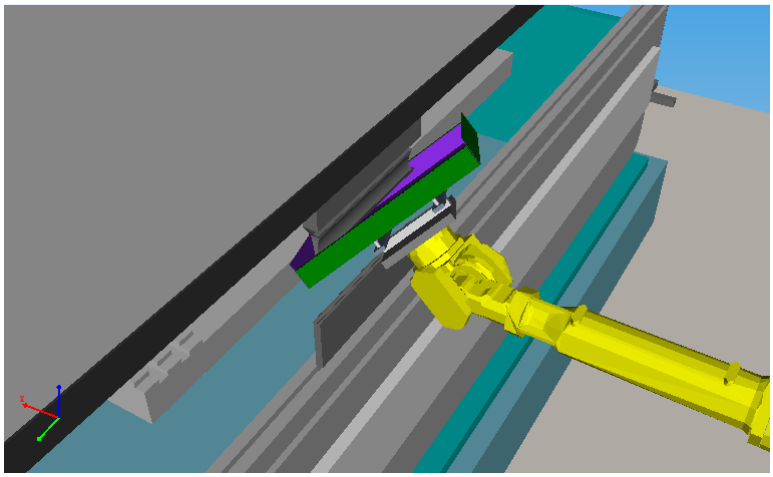

Figura 5.4: Posizione risultate dopo esecuzione operazione **fase 4**

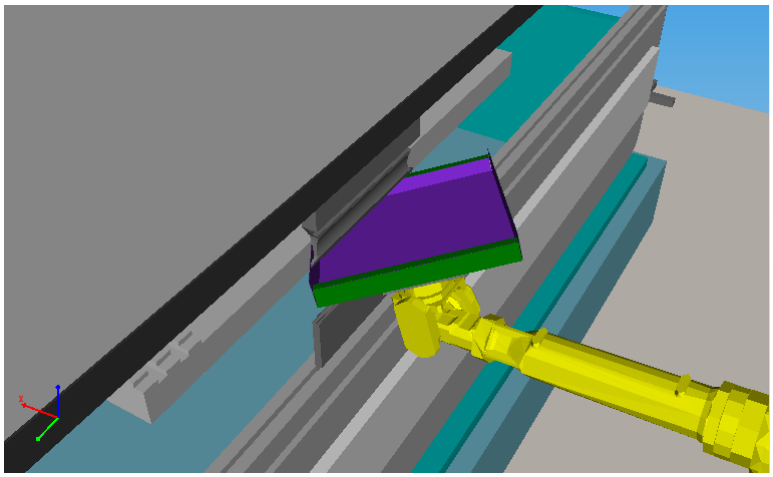

Figura 5.5: Posizione risultate dopo esecuzione operazione **fase 2**

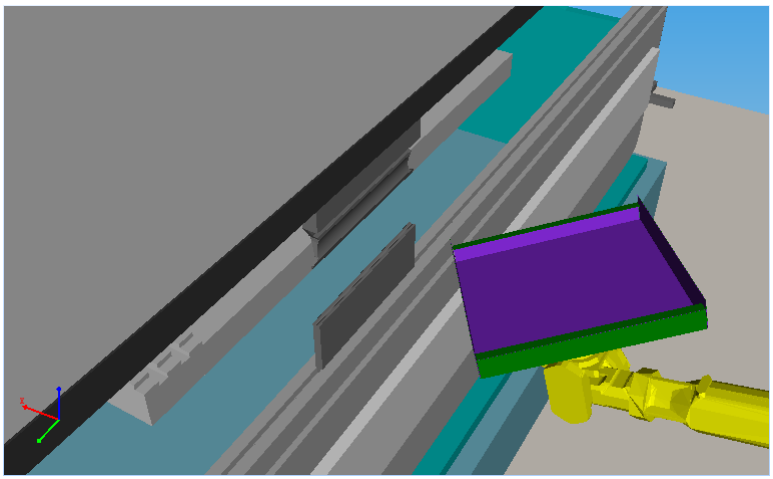

Figura 5.6: Posizione risultate dopo esecuzione operazione **ExtractSimple**

Capitolo 6

Conclusioni

In questa tesi è stato realizzato un software di pianificazione, pensato appositamente per risolvere il problema dell'estrazione di pannelli di lamiera dalla pressa. Il programma realizzato è una parte del simulatore di celle industriali automatiche per la piegatura di pannelli di lamiera, SimulEasy2 della IT-Robotics. Il software è stato scritto in C++.

Nelle prime fasi sono stati condotti degli studi per capire come approcciare il problema e quali metodi potessero essere utilizzati. Dopo diverse prove si è capito che gli algoritmi di pianificazione generali non riuscivano a produrre risultati corretti nelle situazioni più difficili e dovano essere opportunamente configurati.

Nel corso della tesi sono state realizzate alcune versioni del software per provare diverse strade allo scopo di trovare un algoritmo adatto a risolvere il problema. Dopo svariati tentativi, si è scelto di adoperare un approccio che prova iterativamente a muovere l'elaborato lungo diverse direzioni. Ad ogni iterazione le distanze degli spostamenti sono corretti, in modo tale da adattare le singole operazioni ai diversi casi generati. Quando si è riusciti a trovare una nuova posizione valida dell'elaborato, diversa dalle altre, si è cercato di estrarlo direttamente. L'algoritmo ha dato ottimi risultati riuscendo a trovare le soluzioni anche nei casi più complessi. Oltre a questo risultato si è riusciti a sviluppare un algoritmo che non necessita una fase di

configurazione, riuscendo dopo alcune iterazioni a trovare i valori più idonei.

Futuri sviluppi potrebbero riguardare l'efficienza del programma, cercando di rendere l'algoritmo ancora più veloce. Il programma, invece, necessità di una ulteriore fase di progettazione per meglio adattarlo alle restanti libreria del simulatore.

Bibliografia

[1] Stefano Carpin and Gianluigi Pillonetto. Motion planning using adaptive random walks. In *IEEE TRANSACTIONS ON ROBOTICS*, pages 3809–3814, 2005.

[2] Steven M. Lavalle. Rapidly-exploring random trees: A new tool for path planning. Technical report, 1998.

[3] Marius Muja and David G. Lowe. Fast approximate nearest neighbors with automatic algorithm configuration. In *In VISAPP International Conference on Computer Vision Theory and Applications*, pages 331–340, 2009.

[4] John H. Reif. Complexity of the mover's problem and generalizations. In *Proceedings of the 20th Annual Symposium on Foundations of Computer Science*, SFCS '79, pages 421–427, Washington, DC, USA, 1979. IEEE Computer Society.