



UNIVERSITÀ DEGLI STUDI DELLA CALABRIA
DIPARTIMENTO DI INGEGNERIA INFORMATICA, MODELLISTICA,
ELETTRONICA E SISTEMISTICA

CORSO DI LAUREA MAGISTRALE IN INGEGNERIA INFORMATICA

Progetto di
Artificial Intelligence

Professori

Prof. Scarcello Francesco
Ing. Antonio Bono

Studenti

Faiella Vittorio
Gugliotta Carmelo
Pecora Carmelo

Anno Accademico 2023-2024

Indice

1	Modelling	2
1.1	Descrizione del dominio	2
1.2	Descrizione istanze	7
1.2.1	Istanza 1	7
1.2.2	Istanza 2	8
1.2.3	Istanza 3	9
2	Classical Planning	11
2.1	Classi Predicate e Argument	11
2.2	Classe Esl Heuristic	12
2.2.1	Pruning dell'albero di ricerca: isWorth	14
2.2.2	Euristica	16
2.2.3	Istanza 1	18
2.2.4	Istanza 2	18
2.2.5	Istanza 3	19
2.2.6	Riepilogo	21
3	Temporal Planning & Robotics	22
3.1	Modifica del dominio	22
3.2	Modifica delle istanze	24
3.3	Plan Utils	25
3.4	PlanSys2	26
3.4.1	Esecuzione istanza su PlanSys2	27
4	Organizzazione dell'archivio consegnato	29

Capitolo 1

Modelling

Il lavoro progettuale è volto alla risoluzione di un problema di logistica che consiste nel pianificare nel miglior modo possibile i percorsi che gli agenti dovranno compiere per consegnare i beni di cui necessitano alcuni soggetti. Per ottenere una modellazione di questo problema è necessario definire un dominio. Il dominio è un componente fondamentale nella trattazione del suddetto problema perché ci consente di definire i tipi, su cui si dovrà lavorare nel corso della risoluzione del problema, i predicati e le azioni. Esiste anche la sezione requirements in cui vengono importate nel nostro dominio delle librerie che ci consentono di operare.

1.1 Descrizione del dominio

I tipi definiti, elencati di seguito, ci servono per ottenere la rappresentazione dell'ambiente in cui l'agente agirà

```
(:types
  slot ; * A space in a dimensionable object
  box ; * A box that can be filled with content
  carrier ; * A carrier that can be used to transport boxes
  robot ; * A robot that can move around and interact with boxes, carrier and
    contents
  location ; * A place
  content ; * A type of object that can be placed in a box or given to a person
  person ; * A person who can receive content
)
```

- **Slot:** Lo slot è un tipo che ci consente di rappresentare il concetto di spazio sul carrello. Definendo più slot nelle istanze dei problemi sarà possibile inserire più oggetti su uno stesso carrello.
- **Box:** È un oggetto che può contenere un solo bene tra quelli a disposizione.
- **Carrier:** È l'oggetto carrello che una volta istanziato potrà essere usato dall'agente per spostare le box da un luogo ad un altro.
- **Robot:** Il tipo robot rappresenta l'agente che materialmente provvederà a trasportare i beni servendosi degli oggetti definiti in precedenza.
- **Location:** È un oggetto che serve per modellare il concetto di posizione in modo da poter esplicitare il fatto che una persona si trovi in un posto differente rispetto ad un'altra persona o rispetto al robot.
- **Content:** Rappresenta ciò che può essere inserito nelle box e consegnato alle persone.
- **Person:** È un oggetto che serve a rappresentare le persone bisognose di alcuni content.

In seguito, è presente la definizione dei predicati, i quali ci consentono di effettuare delle interrogazioni sulla base di conoscenza e di ottenere delle risposte binarie (true o false). Grazie ai responsi offerti dai predicati ci è possibile verificare se una determinata azione può essere eseguita o meno sulla base delle condizioni imposte dal problema.

```

(:predicates

  (at ?o - object ?l - location) ; object ?o is at location ?l
  (depot-at ?l - location) ; A depot is present at location ?l


  (full ?b - box) ; A box ?b that is full.
  (already-taken ?b) ; box ?b it is not available
  (has-inside ?b - box ?elem - content ) ; box ?b has content ?elem
  (on-carrier ?b - box ?c - carrier) ; box ?b is on carrier ?c


  (empty ?s - slot ?c - carrier) ; A slot ?s of a carrier that is empty.


  (has-content ?p - person ?elem - content ) ; person ?p has content ?elem


  (satisfied-with-at-least-one ?p - person ?elem1 - content ?elem2 - content)
  ; person ?p is satisfied with at least one of the objects ?elem1 or ?elem2
  ; allows to handle the 'or' in the goal


  (is-holding ?r -robot ?c - carrier) ; robot ?r is holding carrier ?c
)

```

- **At:** ci consente di sapere se un certo oggetto o si trova nella posizione “l”.
- **Depot-at:** è un predicato che serve per capire se nella posizione “l” è presente un deposito.
- **Full:** il predicato full applicato ad una box “b” ci permette di sapere se la box è piena o no.
- **Already-taken:** prima di prendere una box, un robot dovrebbe assicurarsi che essa non sia già stata presa da qualche altro agente. Se questo predicato restituisce falso significa che la box non è mai stata presa da nessuno e può quindi essere caricata sul carrello.
- **Has-inside:** ci chiediamo se la box “b” contenga un certo content.
- **On-carrier:** è un predicato che ci permette di sapere se una certa box “b” si trova su un carrello “c”.
- **Empty:** prima di caricare una box sul carrello bisogna essere certi che ci sia ancora almeno uno spazio libero. Empty ci permette di assicurarci che questa condizione sia verificata.
- **Has-content:** tale predicato ci è utile per sapere se una certa persona ha già ricevuto un determinato content
- **Satisfied-with-at-least-one:** questo predicato è necessario per rispettare il vincolo di or nella risoluzione della seconda e terza istanza della sezione “classical planning”. In quell’istanza per il raggiungimento del goal la persona p1 deve avere “food or medicine”; per rispettare questo vincolo ci siamo serviti di questo predicato che ci restituisce true quando almeno uno dei due content è stato ricevuto dalla persona p.
- **Is-holding:** Ogni carrello può essere gestito da un solo robot, pertanto, grazie a questo predicato possiamo essere certi che un robot utilizzerà soltanto carrelli non usati da altri.

Infine, troviamo la definizione delle azioni; esse consentono al robot di interagire con l’ambiente circostante e di effettuare le operazioni necessarie per giungere al soddisfacimento del goal. Un’azione è costituita dai parametri, dalle precondizioni e dagli effetti. I parametri sono gli oggetti in input necessari per il conseguimento dell’obiettivo finale dell’azione. Le precondizioni sono delle condizioni booleane da valutare prima di eseguire l’azione; sono necessarie per valutare se una certa operazione può essere eseguita in un determinato stato. Gli effetti sono le conseguenze dell’esecuzione di un’azione: ciò che è contenuto nella clausola effect induce una modifica sulla base di conoscenza e condiziona le successive interrogazioni di uno o più predicati.

- **Fill-Box:**

```
(:action fill-box
  :parameters ( ?r - robot ?b - box ?elem - content ?l - location )
  :precondition (and
    (not (full ?b)) (depot-at ?l)
    (at ?r ?l) (at ?elem ?l) (at ?b ?l)
  )
  :effect (and (full ?b) (has-inside ?b ?elem))
)
```

L'operazione di fill-box è stata creata per dare la possibilità al robot di riempire una box con un certo contenuto. Per effettuare questa operazione è necessario, come esplicitato nelle precondizioni, che la box non sia piena e che il robot, la box e il content si trovino al deposito. In base alla traccia, infatti, i beni distribuibili si trovano nel deposito pertanto è importante imporre che l'operazione di riempimento della box possa essere effettuata solo se box, robot ed elemento si trovano in prossimità di un deposito. Non avendo specificato alcuna precondizione che impedisca al robot di riempire una box già caricata sul carrello abbiamo supposto che sia possibile farlo. Una volta riempita la box essa risulterà piena pertanto è importante specificare tale condizione nella base di conoscenza. La clausola effect ci consente di esprimere la condizione di pienezza della box "b" e anche di specificare quale contenuto sia stato inserito

- **Unfill-box**

```
(:action unfill-box
  :parameters (?r - robot ?b - box ?elem - content ?l - location)
  :precondition (and
    (full ?b) (has-inside ?b ?elem)
    (depot-at ?l) (at ?r ?l) (at ?b ?l)
  )
  :effect (and (not (full ?b)) (not (has-inside ?b ?elem)))
)
```

Tale azione ci consente di svuotare una box; potremmo definirla come l'azione duale rispetto alla fill-box. Per essere eseguita c'è bisogno di verificare che la box sia piena e che contenga il content denominato "elem". Inoltre, bisogna assicurarsi di essere al deposito; quest'azione, infatti, potrebbe essere eseguita qualora il robot debba sostituire il content di una box con un altro content. È importante, quindi, imporre di essere in prossimità del deposito in modo da poter effettivamente caricare un altro content. La conseguenza di quest'azione è che la box non contiene più il content precedente.

- **Give-content:**

```
(:action give-content
  :parameters (?r - robot ?p - person ?elem - content ?b - box ?l -
    location
  )
  :precondition (and (at ?p ?l) (at ?b ?l) (at ?r ?l)
    (has-inside ?b ?elem) (not (has-content ?p ?elem))
  )
  :effect (and
    (not (has-inside ?b ?elem))
    (not (full ?b)) (has-content ?p ?elem)
  )
)
```

Tramite quest'azione il robot ha la possibilità di consegnare un certo content ad una persona. Ipotizzando che la persona bisognosa del content si trovi in posizione "l" c'è bisogno che la box, con all'interno il content necessario, e il robot si trovino nella stessa posizione. Inoltre, è necessario verificare che la persona non abbia già ricevuto il content. Se le suddette condizioni risultano tutte verificate allora il content viene fornito alla persona, la box viene svuotata e viene segnalato che la box non contiene più il content "elem".

- **Satisfied-with-at-least-one:**

```
(:action satisfied-with-at-least-one
:parameters ( ?p - person ?elem1 - content ?elem2 - content)
:precondition (and
  (not (satisfied-with-at-least-one ?p ?elem1 ?elem2))
  (or
    (has-content ?p ?elem1) (has-content ?p ?elem2)
  )
)
:effect (satisfied-with-at-least-one ?p ?elem1 ?elem2)
)
```

Si tratta di un'azione fittizia introdotta per far fronte alla problematica presentata nelle istanze 2 e 3 del progetto. Per soddisfare le istanze, infatti, bisognerebbe inserire una condizione di or nel goal dei problemi; ciò genera delle problematiche al planner che non consentono di pervenire ad una soluzione. Per poter concludere che è soddisfatta almeno una condizione c'è bisogno che nello stato attuale "satisfied-with-at-least-one" sia ancora falso e che la persona abbia almeno un content tra i due passati in input. Se sono vere le precondizioni allora potremo scrivere nella base di conoscenza che la persona "p" ha ottenuto almeno uno dei due beni denominati "elem1" ed "elem2".

- **Hold-carrier**

```
(:action hold-carrier
:parameters (?r - robot ?c - carrier ?l - location)
:precondition (and (at ?r ?l) (at ?c ?l)
  (not
    (exists (?x - robot)
      (is-holding ?x ?c)
    )
  )
)
:effect (is-holding ?r ?c)
)
```

L'azione ha l'obiettivo di consentire al robot di afferrare un carrello. Prima di far ciò bisogna verificare che il robot e il carrello si trovino nella stessa posizione e che non esista nessun altro robot che abbia già brandito il carrello "c". L'effetto consisterà nell'inserimento nella base di conoscenza di un fatto "is-holding" che impedirà ad ogni altro robot di agire sul carrello "c" afferrato dal robot "r".

- **Release-carrier:**

```
(:action release-carrier
:parameters (?r - robot ?c - carrier ?l - location)
:precondition (and (is-holding ?r ?c) (depot-at ?l)
  (not
    (exists (?b - box)
      (on-carrier ?b ?c)
    )
  )
)
:effect (and (not (is-holding ?r ?c)) (at ?c ?l))
)
```

Dopo aver completato la consegna dei beni alle persone bisognose il carrello può essere rilasciato. L'azione "release-carrier" consente al robot "r" di rilasciare il carrello che precedentemente aveva preso. Prima di rilasciarlo bisogna verificare che il carrello fosse effettivamente gestito dal robot "r", che ci si trovi al deposito e che il carrello sia vuoto (nessuna cassa su di esso). Il carrello deve trovarsi al deposito poiché, presumibilmente, verrà riutilizzato da qualche altro agente per caricare altre casse. Inoltre, bisogna assicurarsi che sia vuoto affinché non vi siano ancora consegne in sospeso.

- **Load-carrier:**

```

(:action load-carrier
  :parameters (?r - robot ?b -box ?c - carrier ?s - slot ?l - location)
  :precondition (and
    (is-holding ?r ?c)
    (at ?b ?l) (at ?r ?l) (at ?c ?l)
    (not (already-taken ?b))
    (empty ?s ?c)
  )
  :effect (and (on-carrier ?b ?c) (already-taken ?b)
    (not (empty ?s ?c))
  )
)

```

Quest'azione consente al robot di caricare una cassa (vuota o piena) sul carrello. Affinché sia possibile effettuare quest'operazione c'è bisogno di verificare che il robot abbia preso il carrello in questione. Inoltre, la box, il robot e il carrello devono trovarsi nella stessa posizione, deve esserci almeno uno slot libero sul carrello e la box non deve essere stata già presa da qualche altro robot. Se le condizioni sono vere allora la box verrà caricata sul carrello, risulterà "impegnata" e lo slot "s" verrà dichiarato pieno.

- **Unload-carrier:**

```

(:action unload-carrier
  :parameters (?r - robot ?b - box ?c - carrier ?s - slot ?l -
    location)
  :precondition (and
    (is-holding ?r ?c)
    (at ?r ?l) (at ?c ?l) (depot-at ?l)
    (on-carrier ?b ?c) (not (full ?b))
    (not (empty ?s ?c) )
  )
  :effect (and (not (on-carrier ?b ?c)) (at ?b ?l)
    (empty ?s ?c)
  )
)

```

L'operazione di unload consente di scaricare una cassa dal carrello "c". Per far ciò è necessario che il carrello sia effettivamente afferrato dal robot "r", che la box vuota sia sul carrello e occupi uno slot. Inoltre, abbiamo imposto che l'operazione debba svolgersi al deposito in modo tale da lasciare lì le casse vuote a disposizione di altri agenti. Gli effetti consistono nella liberazione di uno slot sul carrello "c", nel fatto che la box si trovi al deposito e non più sul carrello.

- **Move:**

```

(:action move
  :parameters (?r - robot ?from ?to - location)
  :precondition (and (at ?r ?from) (not (at ?r ?to))
    (not
      (exists (?x - carrier)
        (is-holding ?r ?x)
      )
    ))
  :effect (and (not (at ?r ?from)) (at ?r ?to))
)

```

L'azione "move" è stata concepita per consentire al robot di spostarsi da una posizione all'altra. Tra i parametri, oltre al robot, troviamo la posizione di partenza e la posizione di arrivo; per poter eseguire questa operazione è necessario che il robot si trovi nella posizione di partenza e che non sia già nella posizione di arrivo. Inoltre, è necessario che il robot non stia gestendo alcun carrello. L'ultima condizione ci consente

di collegare quest'azione solo allo spostamento di un robot, evitando che un robot con carrello possa essere spostato lasciando il carrello nella posizione “from”. Una volta valutate le precondizioni il robot si sposterà e verrà aggiornata la base di conoscenza.

- **Move-carrier:**

```
(:action move-carrier
:parameters (?r - robot ?from ?to -location ?c - carrier)
:precondition (and (at ?r ?from) (not(at ?r ?to)) (at ?c ?from)
(is-holding ?r ?c))
:effect (and (not (at ?r ?from)) (not (at ?c ?from))
(at ?r ?to) (at ?c ?to)
(forall (?b - box)
(when (on-carrier ?b ?c)
(and (not (at ?b ?from)) (at ?b ?to)
))
))
)
)
```

L'azione in questione consente lo spostamento del robot e del suo carrello da una posizione “from” ad una posizione “to”. Per poter eseguire quest'operazione è necessario che il robot e il carrello si trovino nella posizione di partenza e non siano nella posizione di arrivo. Per evitare di spostare un carrello assegnato ad un altro robot si usa il predicato “is-holding”. L'esecuzione dell'azione comporta lo spostamento del robot “r”, del carrello “c” e di tutte le box caricate su “c”. Per spostare le scatole si effettua un ciclo su tutte le box presenti nella base di conoscenza e, tramite la valutazione del predicato “on-carrier”, si procede allo spostamento delle box che nello stato corrente sono caricate sul carrello in questione.

1.2 Descrizione istanze

Le richieste progettuali si articolano in tre istanze di problemi ognuna delle quali ha una configurazione iniziale da cui partire e un goal da raggiungere.

1.2.1 Istanza 1

Nella prima istanza disponiamo di cinque scatole (“box”) collocate nel deposito e una quantità di beni sufficiente a soddisfare le esigenze di tutte le persone. In questo caso i soggetti da soddisfare sono tre: due si trovano nello stesso punto dello spazio mentre il terzo si trova in una posizione differente. Per giungere al soddisfacimento di tutte le richieste avremo a disposizione un solo agente e un solo carrello su cui sarà possibile caricare al più quattro box. L'obiettivo finale è quello di consegnare a tutte le persone ciò di cui hanno bisogno: la persona p1 ha bisogno di cibo e medicine, la persona p2 (nella stessa posizione di p1) ha bisogno di medicine e la persona p3 ha bisogno di cibo. Di seguito si riporta il codice utilizzato per la definizione del problema:

```
(define (problem es1-istanza-one)

(:domain emergency-services-logistics)
(:objects
r1 - robot
p1 p2 p3 - person
b1 b2 b3 b4 b5 - box
depot l1 l2 - location
ca - carrier
s1-ca s2-ca s3-ca s4-ca - slot
food - content
medicine - content
)

(:init
(depot-at depot)
(at r1 depot) (at ca depot)
(at b1 depot) (at b3 depot) (at b4 depot) (at b5 depot)
(at food depot) (at medicine depot)
```



```

        (at p1 11) (at p2 11) (at p3 12)
        (empty s1-ca ca) (empty s2-ca ca) (empty s3-ca ca) (empty s4-ca ca)
    )

;; The task is to provide people with the content they need
(:goal
  (and
    (has-content p1 food)
    (has-content p1 medicine)
    (has-content p2 medicine)
    (has-content p3 food)
  )
)
)

```

La clausola `problem` è seguita dal nome che abbiamo attribuito al problema mentre nella clausola `domain` abbiamo riportato lo stesso nome specificato nel file di dominio precedentemente descritto. Grazie alla clausola `objects` possiamo istanziare, nel rispetto dei vincoli imposti dal problema, gli oggetti che il planner avrà a disposizione per giungere al goal. Nella clausola `init` definiamo lo stato iniziale, inserendo nella base di conoscenza i fatti necessari a capire quale sia la configurazione di partenza; carrello, robot, scatole e beni vengono posizionate nel deposito mentre le persone vengono posizionate in punti dello spazio diversi dal deposito. Inoltre, i quattro slot definiti in precedenza vengono dichiarati vuoti e pronti al caricamento delle box. Infine, la clausola `goal` ci consente di specificare lo stato che vogliamo raggiungere: le persone devono ricevere i beni che sono specificati nella definizione dell'istanza.

1.2.2 Istanza 2

La seconda istanza presenta alcune differenze rispetto alla precedente. In questo caso abbiamo a disposizione due carrelli, ognuno con una capacità massima pari a due, due robot e due scatole. Le persone da soddisfare sono sei due delle quali, `p1` e `p2`, si trovano nuovamente nella stessa posizione. Per raggiungere il goal c'è bisogno che vengano soddisfatti i bisogni di tutte le persone; la persona `p1` ha bisogno almeno di una risorsa tra `food` e `tools` il che introduce una problematica per il raggiungimento del goal. Tale problema è stato risolto definendo l'azione fittizia `"satisfied-with-at-least-one"` e l'omonimo predicato.

```

(define (problem es1-istanza-two)

  (:domain emergency-services-logistics)

  (:objects

    p1 p2 p3 p4 p5 p6 - person
    b1 b2 b3 - box
    depot 11 12 13 14 15 - location
    ca cb - carrier
    s1-ca s2-ca s1-cb s2-cb - slot
    food - content
    medicine - content
    tool - content
    r1 r2 - robot

  )

  (:init

    (depot-at depot)

    (at r1 depot) (at r2 depot);robot one and robot two are at the same place
    (Depot)
    (at ca depot) (at cb depot) ;carrier one and carrier two are at the same place
    (Depot)
  )
)

```

```

(at b1 depot) (at b2 depot) (at b3 depot) ;box one box 2 and box 3 are at the
same place (Depot)
(at food depot) (at medicine depot) (at tool depot) ;food medicine and tools
are at the same place (Depot)
(at p1 l1) (at p2 l1) (at p3 l2) (at p4 l3) (at p5 l4) (at p6 l5) ; person one
and person two are at location one. Other people are at different place.

(empty s1-ca ca) (empty s2-ca ca) (empty s1-cb cb) (empty s2-cb cb) ; In this
case carrier one and also carrier two have two slot.

)

(:goal

  (and
    (satisfied-with-at-least-one p1 food tool)
    (has-content p2 medicine)
    (has-content p3 medicine)

    (has-content p4 medicine)
    (has-content p4 food)

    (has-content p5 food)
    (has-content p5 medicine)
    (has-content p5 tool)

    (has-content p6 food)
    (has-content p6 medicine)
    (has-content p6 tool)
  )

)
)

```

1.2.3 Istanza 3

L'istanza tre è molto simile alla precedente eccetto per due condizioni: le scatole a nostra disposizione sono quattro e le persone da soddisfare sono otto. Di seguito la definizione:

```

(define (problem es1-istance-three)

  (:domain emergency-services-logistics)

  (:objects

    p1 p2 p3 p4 p5 p6 p7 p8 - person
    b1 b2 b3 b4 - box
    depot l1 l2 l3 l4 l5 l6 l7 - location
    ca cb - carrier
    s1-ca s2-ca s1-cb s2-cb - slot
    food - content

    medicine - content

    tool - content
    r1 r2 - robot

  )
)

```

```

(:init

  (depot-at depot)

  (at r1 depot) (at r2 depot) ;robot one and robot two are at the same place
  (Depot)
  (at ca depot) (at cb depot) ;carrier one and carrier two are at the same
  place (Depot)
  (at b1 depot) (at b2 depot) (at b3 depot) (at b4 depot);box one box 2 and
  box 3 are at the same place (Depot)
  (at food depot) (at medicine depot) (at tool depot) ;food medicine and
  tools are at the same place (Depot)
  (at p1 11) (at p2 11) (at p3 12) (at p4 13) (at p5 14) (at p6 15) (at p7
  16) (at p8 17) ; person one and person two are at location one. Other
  people are at different place.
  (empty s1-ca ca) (empty s2-ca ca) (empty s1-cb cb) (empty s2-cb cb) ; In
  this case carrier one and also carrier two have two slot.

)

(:goal

  (and
    (satisfied-with-at-least-one p1 food tool)
    (has-content p2 medicine)

    (has-content p3 medicine)

    (has-content p4 medicine)
    (has-content p4 food)

    (has-content p5 food)
    (has-content p5 medicine)
    (has-content p5 tool)

    (has-content p6 food)
    (has-content p6 medicine)
    (has-content p6 tool)

    (has-content p7 food)
    (has-content p7 medicine)
    (has-content p7 tool)

    (has-content p8 food)
    (has-content p8 medicine)
    (has-content p8 tool)
  )
)
)

```

Capitolo 2

Classical Planning

Il capitolo attuale assume una posizione centrale nell'ambito della soluzione delle istanze proposte, avvalendosi della libreria PDDL4J. Questa libreria Java, fornisce una serie di classi per la manipolazione e la risoluzione di problemi di pianificazione espressi nel linguaggio PDDL v1.2 (Planning Domain Definition Language).

In PDDL4J, i problemi (istanze) sono delineati da una lista di Fluent, dove ciascun Fluent rappresenta una proposizione il cui valore di verità (True/False) evolve nel corso del processo di pianificazione. In un contesto di pianificazione classica, ogni nodo nell'albero di ricerca è composto da una serie di predicati veri che riflettono lo stato attuale dell'ambiente.

Per semplificare e rendere più intuitiva la valutazione degli stati dell'ambiente, sono state implementate le classi *Esl Heuristic*, *Predicate* e *Argument*. Queste classi offrono un'astrazione di alto livello per comprendere lo stato dell'ambiente e facilitano la valutazione degli stati esplorati, oltre a consentire eventuali operazioni di potatura dell'albero di ricerca nel processo di pianificazione.

2.1 Classi Predicate e Argument

La classe **Predicate** rappresenta i predicati che descrivono aspetti specifici dello stato dell'ambiente. Nel contesto di pianificazione, tali predicati riflettono le condizioni che possono variare durante l'esecuzione delle azioni. La classe **Argument**, invece, offre una rappresentazione chiara e semplificata dei singoli atomi coinvolti in un predicato, contribuendo a una valutazione più intuitiva dello stato dell'ambiente.

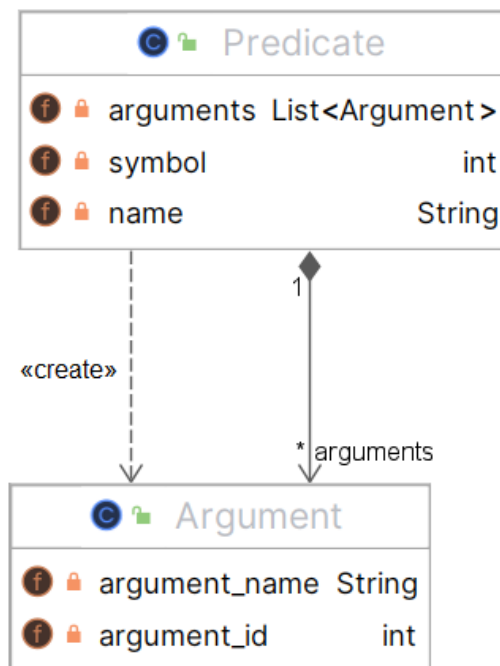


Figura 2.1: Predicate.java and Argument.java

Di seguito si riporta la descrizione di ogni campo (field) delle classi *Predicate* e *Argument*

- **Predicate**

- **symbol** (*int*): Rappresenta l'identificatore univoco del predicato.
- **name** (*String*): Rappresenta il nome del predicato. Ad esempio, se il predicato rappresenta la condizione "depot-at", questo campo conterrà la stringa "depot-at".
- **arguments** (*List* < *Argument* >): È una lista di oggetti **Argument** che rappresentano gli argomenti coinvolti nel predicato. Ogni argomento ha un identificatore e un nome.

- **Argument**

- **argument_id** (*int*): Rappresenta l'identificatore univoco dell'argomento.
- **argument_name** (*String*): Rappresenta il nome dell'argomento. Ad esempio, se l'argomento è denominato "r1", questo campo conterrà la stringa "r1".

Entrambe le classi sono progettate per offrire una struttura dati modulare e organizzata per la gestione delle informazioni relative alle condizioni di stato nel dominio della pianificazione.

2.2 Classe Esl Heuristic

La classe *EslHeuristic* assume un ruolo centrale nell'implementazione di un'euristica dedicata al dominio ESL (Emergency Services Logistic) all'interno del contesto di pianificazione classica. La sua progettazione è orientata a fornire valutazioni intelligenti dello stato e tecniche di potatura per ottimizzare l'esecuzione dell'algoritmo di ricerca (Es. A*) nei problemi di pianificazione specifici del dominio ESL.

EslHeuristic	
NAME	String
instance	EslHeuristic
problem	Problem
idConditionMap	Map<Integer, Predicate>
typeToArguments	Map<String, List<Argument>>
heuristic	StateHeuristic
carrierInfo	Map<String, Integer>
getPredicates(int[])	List<Predicate>
getInstance (Problem, Name)	EslHeuristic
estimate(State, Condition)	int
estimate(Node, Condition)	double
isWorth(Node, Action)	boolean
createIdConditionMap()	void
evaluateMoveCarrier(Node, Action)	boolean
setUpVariableOfTheProblem()	void
createTypeToArguments ()	void
evaluateGiveContent (Node, Action)	boolean
evaluateFillBox(Node, Action)	boolean
evaluateSatisfiedWithAtLeastOne (Node, Action)	boolean
checkBoxes(List<Predicate>, List<Predicate>)	int
evaluateMove(Node, Action)	boolean
getPositionsToReach (List<Predicate>, List<Predicate>)	int

Figura 2.2: EslHeuristic.java

La classe *EslHeuristic* estende la classe *RelaxedGraphHeuristic* del framework PDDL4J ed è implementata come un Singleton, un design pattern che assicura che una sola istanza della classe esista nell'intero sistema. Questo significa che la classe può essere istanziata solo una volta, e successivamente, tutte le richieste di ottenere un'istanza restituiranno sempre la stessa istanza creata inizialmente.

L'implementazione del Singleton è realizzata attraverso l'utilizzo di un campo statico *instance* e il metodo *getInstance* che controlla se un'istanza è già stata creata. Se l'istanza non esiste, il metodo la crea in modo sincronizzato all'interno di un blocco *synchronized*, garantendo che solo un'istanza della classe *EslHeuristic* esista nell'applicazione.

Di seguito si riporta una breve descrizione dei campi (field) della classe *EslHeuristic*:

- Campi **problem** e **heuristic**: I campi *problem* e *heuristic* sono dichiarati come finali e contengono rispettivamente l'oggetto *Problem*, rappresentante il problema di pianificazione associato, e un eventuale euristica di default del framwork PDDL4J (*AJUSTED_SUM*, *AJUSTED_SUM2*, *FAST_FORWARD* ...)
- Mappe **idConditionMap** e **typeToArguments**: Le mappe *idConditionMap* e *typeToArguments* svolgono un ruolo fondamentale nella traduzione e nell'associazione delle condizioni del problema

$$idConditionMap : \{ \langle id_1 : predicate_1 \rangle, \dots, \langle id_n : predicate_n \rangle \}$$

$$typeToArguments : \{ \langle typeName_1 : [Arg_1, \dots, Arg_n] \rangle, \dots, \langle typeName_n : [Arg_1, \dots, Arg_n] \rangle \}$$

La prima Mappa è stata implementata per fornire un modo comodo per accedere e interpretare le condizioni che descrivono lo stato di un nodo all'interno dell'albero di ricerca. Infatti si osserva che:

- Ogni nodo all'interno dell'albero di ricerca è rappresentato come un'istanza della classe **Node** fornita dal framework PDDL4J. Questa classe estende **State**, che a sua volta è una sottoclasse di **BitVector**. L'utilizzo di questa gerarchia di classi è fondamentale per rappresentare lo stato corrente del problema di pianificazione in un formato efficiente e gestibile.

Nel dettaglio, quando effettuiamo un'operazione come *stream.toArray()* su un oggetto di tipo *State*, otteniamo un vettore contenente gli ID dei predicati che descrivono lo stato corrente. Questi ID, a livello più basso, rappresentano in modo efficace la configurazione specifica dello stato.

Per rendere questa rappresentazione più chiara e interpretabile dal punto di vista della programmazione, facciamo uso della mappa *idConditionMap*. Questa mappa stabilisce una correlazione tra ciascun ID di predicato e un oggetto più esplicito e facilmente comprensibile, offrendo così una descrizione più astratta dello stato attuale. In pratica, richiamiamo il metodo *getPredicates(int[] state)* e passiamo come argomento il vettore contenente gli ID dei predicati che descrivono lo stato esplorato. Successivamente, consultando la mappa *idConditionMap*, traduciamo gli ID dei predicati in informazioni significative sulle condizioni presenti nello stato, semplificando così la comprensione e la manipolazione di tali informazioni all'interno dell'algoritmo di ricerca.

La seconda Mappa associa il tipo di argomento a una lista di argomenti istanziati nel problema, offrendo un'organizzazione chiara delle relazioni tra tipi e istanze. Ad esempio, potrebbe apparire come:

$$\{ \langle robot : [r1, r2] \rangle, \dots, \langle carrier : [ca, cb] \rangle \}$$

- Mappa **carrierInfo**: La mappa *carrierInfo* contiene informazioni sullo spazio disponibile di ciascun trasportatore, associando il nome unico del trasportatore a un valore numerico che indica la disponibilità di spazio. Ad esempio:

$$\{ \langle ca : 2 \rangle, \dots, \langle cb : 2 \rangle \}$$

Successivamente, sono stati dichiarati due metodi chiave: **estimate** e **isWorth**. Il primo, *estimate*, è progettato per stimare la distanza dal goal, offrendo una valutazione dell'idoneità di uno stato nel contesto della ricerca della soluzione ottimale. Il secondo, *isWorth*, svolge un ruolo cruciale nel determinare quali azioni dovrebbero essere considerate durante l'esplorazione degli stati successivi. Riconosce che non tutte le azioni potrebbero contribuire in modo significativo al raggiungimento del goal e, di conseguenza, decide di eliminare quelle che potrebbero rallentare il processo di ricerca. Questo approccio mira a ottimizzare l'esplorazione dell'albero di ricerca, concentrando l'attenzione sulle azioni più rilevanti per il raggiungimento del goal in modo efficiente.

Dettagli più approfonditi su entrambi i metodi verranno forniti nella prossima sezione, dove saranno analizzati in modo più dettagliato per comprendere appieno il loro ruolo nella strategia di ricerca implementata.

2.2.1 Pruning dell'albero di ricerca: isWorth

Nel contesto dell'ottimizzazione del processo di ricerca, abbiamo introdotto una significativa modifica agli algoritmi di ricerca A* ed Enforced Hill Climbing. Questa modifica mira a rendere l'esplorazione degli stati più efficiente, focalizzandosi non solo sulle azioni tecnicamente possibili rispetto allo stato corrente, ma anche su quelle che contribuiscono in modo significativo al raggiungimento del goal.

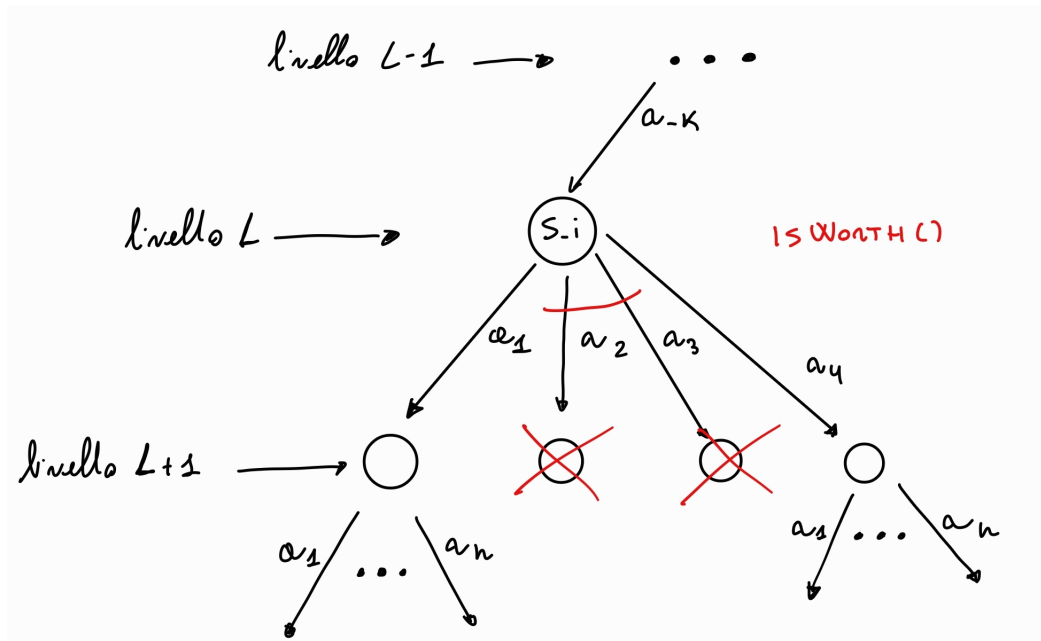


Figura 2.3: Idea alla base di isWorth

Il fulcro di questa innovazione è rappresentato dal metodo `isWorth` implementato nella classe `EslHeuristic`. Durante l'esecuzione degli algoritmi di ricerca, il metodo `isWorth` viene chiamato per ogni azione possibile a partire da uno stato specifico. Esso valutando attentamente la rilevanza di ciascuna azione, permette di effettuare un pruning dell'albero di ricerca, eliminando le azioni che, sebbene tecnicamente possibili, non contribuiscono in modo significativo al progresso verso il raggiungimento del goal. Questa ottimizzazione si traduce in una maggiore efficienza computazionale e in una focalizzazione mirata sull'esplorazione delle azioni più promettenti per raggiungere l'obiettivo finale.

Listing 2.1: Il metodo `isWorth` che valuta se un'azione è rilevante.

```
public boolean isWorth(Node current, Action op){

    String action_name=op.getName();

    switch (action_name){
        case "move":
            return evaluateMove(current,op);
        case "move-carrier":
            return evaluateMoveCarrier(current,op);
        case "fill-box":
            return evaluateFillBox(current,op);
        case "give-content":
            return evaluateGiveContent(current,op);
        case "satisfied-with-at-least-one":
            return evaluateSatisfiedWithAtLeastOne(current,op);
        default:
            return true;
    }
}
```

Prima di esaminare il processo di valutazione di ciascuna azione, è opportuno comprendere che nella libreria PDDL4J, ogni oggetto **Action** è definito da due vettori di interi: *parameters* e *instantiations*. Il vettore *instan-*

tations contiene gli ID degli oggetti coinvolti nell'azione, nell'ordine in cui sono stati definiti nel file di dominio PDDL. Al contrario, il vettore *parameters* contiene, nelle medesime posizioni, il tipo degli oggetti corrispondenti.

- **Action Move:** Il metodo *evaluateMove* verifica se l'azione di spostamento (move) ha senso nell'ambito dello stato corrente. In particolare, considera la seguente situazione:

- Verifica se nella posizione di destinazione (specificata come parametro nell'azione) esiste un carrello. Questo viene controllato esaminando lo stato corrente e verificando la presenza di un predicato "at" con l'argomento corrispondente alla destinazione. Se un carrello è presente, il metodo restituisce true, indicando che l'azione di spostamento ha senso. Altrimenti restituisce false.

In sintesi, il metodo valuta appropriata la move solo quando il robot può raggiungere un carrello nella destinazione. Questo approccio mira a evitare spostamenti inutili a vuoto.

Si sottolinea che l'esecuzione dell'azione di spostamento (move) è condizionata alla mancanza di un carrello in mano al robot, come specificato nella sua definizione all'interno del dominio. Questa azione è stata concepita con l'obiettivo di consentire all'agente robotico di raggiungere un carrello, ponendo quindi il vincolo che il robot non debba già avere un carrello in mano prima di poter eseguire con successo l'azione di spostamento.

- **Action Move Carrier:** Il metodo *evaluateMoveCarrier* è progettato per valutare l'utilità dell'azione di spostamento di un carrello (move-carrier) in base allo stato corrente e agli obiettivi da raggiungere. Ecco una spiegazione del suo funzionamento:

- Nel contesto in cui l'azione move-carrier viene eseguita verso il deposito, la sua utilità è strettamente condizionata alla presenza di spazio disponibile nel carrello che si sta spostando. In altre parole, questa azione è considerata significativa solo quando ci sono spazi liberi sul carrello o casse vuote da caricare.
- Se il movimento è verso destinazioni diverse dal deposito, il metodo esamina se ci sono persone, nelle posizioni di arrivo che desiderano il contenuto attualmente presente nel carrello. In tal caso, verifica se il carrello contiene almeno uno degli elementi richiesti dalle persone. Se questa condizione è soddisfatta, l'azione è considerata utile.

In sintesi, il metodo *evaluateMoveCarrier* valuta attentamente le condizioni sia per il movimento verso il deposito che per il movimento verso destinazioni diverse, garantendo che l'azione di spostamento del carrello sia eseguita in modo significativo e in linea con gli obiettivi dell'istanza in esecuzione.

- **Action Fill Box:** Il metodo *evaluateFillBox* è progettato per valutare se ha senso eseguire l'azione fill-box, ovvero riempire una scatola con un determinato elemento, in base allo stato corrente e agli obiettivi da raggiungere.

- Vengono individuati gli obiettivi già soddisfatti al fine di identificare gli elementi necessari per raggiungere quelli ancora non completati: Si genera una lista di elementi richiesti per soddisfare gli obiettivi che non sono ancora stati raggiunti, escludendo quelli già presenti all'interno delle scatole, poiché non è necessario riempire la scatola con elementi già disponibili.

Infine, si verifica se l'elemento che si desidera aggiungere alla scatola è presente nella lista degli elementi richiesti. Se questa condizione è soddisfatta, l'azione è considerata significativa e utile per il raggiungimento del goal.

In sintesi, *evaluateFillBox* assicura che l'azione di riempire la scatola abbia senso solo se l'elemento da aggiungere contribuisce al raggiungimento degli obiettivi non ancora soddisfatti

- **Action Give Content:** il metodo *evaluateGiveContent* è progettato per valutare se ha senso eseguire l'azione give-content, cioè consegnare un certo contenuto a una specifica persona.

- Il metodo controlla se esiste un obiettivo tra quelli ancora da soddisfare che richiede la consegna dell'elemento specificato alla persona specificata. Se questa condizione è soddisfatta, l'azione è considerata significativa.

In sintesi, *evaluateGiveContent* assicura che l'azione di consegnare un determinato contenuto abbia senso solo se esiste un obiettivo che richiede tale consegna tra quelli ancora non completati.

- **Action SatisfiedWithAtLeastOne:** Il metodo *evaluateSatisfiedWithAtLeastOne* è progettato per valutare se ha senso eseguire l'azione satisfied-with-at-least-one, che esprime la necessità di soddisfare almeno uno tra due contenuti specifici per una persona specifica.

- il metodo controlla se esiste un obiettivo tra quelli ancora da soddisfare che richiede la consegna di almeno uno tra i due elementi specificati per la persona coinvolta. Se questa condizione è soddisfatta, l'azione è considerata significativa e utile ai fini della pianificazione.

In sintesi, *evaluateSatisfiedWithAtLeastOne* assicura che l'azione di consegnare almeno uno tra i due contenuti specificati abbia senso solo se esiste un obiettivo che richiede tale soddisfazione tra quelli ancora non completati.

2.2.2 Euristica

Nel contesto della pianificazione, l'euristica gioca un ruolo cruciale nel guidare gli algoritmi di ricerca verso soluzioni efficienti. L'obiettivo di un'euristica è fornire una stima della distanza tra lo stato corrente e il goal, consentendo all'algoritmo di scegliere percorsi più promettenti durante la ricerca.

L'euristica implementata in questo contesto specifico è classificata come "non ammissibile", il che significa che potrebbe sovrastimare la distanza reale dal goal. Nonostante questa non-ammissibilità, l'euristica è progettata per essere informativa e guidare l'algoritmo nella scoperta di soluzioni ottimali o quasi ottimali in modo più efficiente.

Si esamina di seguito più nel dettaglio l'implementazione e le componenti chiave.

Listing 2.2: Il metodo `estimate` che valuta la lontananza dal goal a partire dallo stato corrente

```
public int estimate(State next, Condition goal){
    super.setAdmissible(false);
    int estimated_value=0;

    List<Predicate> next_state= getPredicates(next.stream().toArray());
    List<Predicate> goals= getPredicates(goal.getPositiveFluents()
        .stream().toArray());

    List<Predicate> goals_already_satisfied=Utility
        .getGoalAlreadySatisfied(next_state,goals);

    List<Predicate> goals_not_satisfied_yet=Utility.
        getGoalNotAlreadySatisfied(goals,goals_already_satisfied);

    //Dobbiamo stimare la lontananza dal goal;
    //Dobbiamo ancora effettuare tante azioni
    //ancora quanti sono i goal da soddisfare
    estimated_value+=goals_not_satisfied_yet.size();

    //Dobbiamo ancora effettuare tante azioni
    //quante sono i posti differenti in cui dobbiamo ancora andare;
    estimated_value+=getPositionsToReach(next_state,goals_not_satisfied_yet);

    //Dobbiamo ancora effettuare tante azioni
    //quanti sono gli item richiesti
    //non ancora caricati nelle casse.
    estimated_value+=checkboxes(next_state,goals_not_satisfied_yet);

    return estimated_value+heuristic.estimate(next,goal);
}
```

Il metodo `estimate` è progettato per valutare la distanza stimata dal goal, incorporando vari aspetti del problema di pianificazione.

- **Obiettivi non ancora soddisfatti:** L'euristica inizia valutando il numero di obiettivi non ancora soddisfatti (`goals_not_satisfied_yet`) nello stato corrente. Il numero di azioni necessarie per soddisfare questi obiettivi contribuisce all'`estimated_value`.

Infatti la somma di `goals_not_satisfied_yet` rappresenta il numero di azioni `give-content` che sono ancora necessarie per soddisfare gli obiettivi. Ogni elemento richiesto e non ancora consegnato richiede un'azione di `give-content`. Pertanto, sommando il numero di obiettivi non ancora soddisfatti, stiamo effettivamente considerando quante azioni `give-content` sono necessarie per coprire tutti gli elementi richiesti.

- **Posizioni ancora da raggiungere:** Viene considerato il numero di posti differenti in cui è necessario ancora recarsi per consegnare i beni alle persone non ancora soddisfatte. Questo valore è aggiunto all'`estimated_value`.

Infatti la somma delle posizioni ancora da raggiungere, attraverso l'uso di `getPositionsToReach`, tiene conto delle azioni `move-carrier` che sono necessarie per raggiungere le diverse posizioni dove sono presenti persone non ancora soddisfatte. Questo è significativo perché prima di effettuare azioni di `give-content` a una specifica persona, è essenziale raggiungerla. Quindi, sommando le posizioni da raggiungere, stiamo valutando quanti spostamenti sono ancora richiesti per soddisfare gli obiettivi.

- **Elementi Richiesti non ancora presenti nelle casse:** Si valutano quanti elementi sono ancora da caricare nelle casse per soddisfare gli obiettivi pendenti e quanti elementi sono stati erroneamente caricati e devono essere rimossi. Questo approccio è finalizzato all'ottimizzazione del contenuto delle casse, **favorendo stati** in cui le casse contengono esclusivamente gli elementi necessari per raggiungere gli obiettivi specifici del problema.

Listing 2.3: Il metodo `checkBox`

```
private int checkBox(List<Predicate> predicates,
    List<Predicate> goalsNotSatisfiedYet) {
    // List of elements contained in the boxes
    List<Argument> boxContents = predicates.stream()
        .filter(p -> "has-inside".equals(p.getName()))
        .map(p -> p.getArguments().get(1))
        .collect(Collectors.toList());

    // List of elements required to achieve the goal
    List<Argument> requiredGoalElements = Utility.
        getListOfElementsRequiredToAchieveTheGoal(goalsNotSatisfiedYet);

    // Number of elements present in requiredGoalElements
    //but not in boxContents: these are the items not yet loaded
    //in the boxes but are needed
    int x1 = (int) requiredGoalElements.stream()
        .filter(arg -> {
            boolean removed = boxContents.remove(arg);
            //Se e' stato rimosso allora era contenuto
            return !removed;
        })
        .count();

    // Number of elements present in boxContents
    //but not in requiredGoalElements: these are the
    //loaded items that are not needed
    int x2 = boxContents.size();

    return (x1 + x2);
}
```

In altre parole, $x1$ rappresenta il conteggio degli elementi che sono necessari per raggiungere gli obiettivi, ma attualmente non sono presenti nelle scatole. D'altra parte, $x2$ rappresenta il conteggio degli elementi presenti nelle scatole che non sono richiesti per raggiungere gli obiettivi. La somma di $x1$ e $x2$ fornisce una stima

del numero totale di azioni necessarie per gestire gli elementi nelle scatole in modo che siano in linea con gli obiettivi del problema.

- **Somma con Euristiche AjustedSum:** Infine, il valore calcolato è sommato al valore stimato dall'euristica AjustedSum (heuristic.estimate(next, goal)), ottenendo una stima complessiva. L'aggiunta di tale valore apporta una raffinatezza supplementare alla stima globale.

In sintesi, il metodo fornisce una valutazione complessiva della distanza dal goal, incorporando diverse metriche e criteri non ammissibili per ottenere una stima più dettagliata e informativa.

Nei metodi in esame, viene utilizzata la classe Utility, una classe di utilità che incorpora vari metodi per la manipolazione e l'estrazione di informazioni da liste. Questo design è stato adottato per evitare l'accumulo di metodi privati all'interno della classe EslHeuristic, preservando così la chiarezza e la leggibilità del codice.

2.2.3 Istanza 1

Il piano generato è il seguente:

```
* Starting ASTAR search with AJUSTED_SUM heuristic and EslHeuristic By Carmelo-Carmelo-Vittorio
* ASTAR search succeeded
```

found plan as follows:

```
00: (          hold-carrier r1 ca depot) [0]
01: (load-carrier r1 b1 ca s4-ca depot) [0]
02: (    fill-box r1 b5 medicine depot) [0]
03: (load-carrier r1 b5 ca s2-ca depot) [0]
04: (    fill-box r1 b1 food depot) [0]
05: (    move-carrier r1 depot l1 ca) [0]
06: (  give-content r1 p1 food b1 l1) [0]
07: (give-content r1 p1 medicine b5 l1) [0]
08: (    move-carrier r1 l1 depot ca) [0]
09: (    fill-box r1 b1 food depot) [0]
10: (    fill-box r1 b5 medicine depot) [0]
11: (    move-carrier r1 depot l1 ca) [0]
12: (give-content r1 p2 medicine b5 l1) [0]
13: (    move-carrier r1 l1 l2 ca) [0]
14: (  give-content r1 p3 food b1 l2) [0]
```

```
time spent:      0,03 seconds parsing
                 0,07 seconds encoding
                 0,17 seconds searching
                 0,27 seconds total time
```

```
memory used:     1,12 MBytes for problem representation
                 0,22 MBytes for searching
                 1,34 MBytes total
```

Il planner è riuscito a trovare un piano ottimale in pochissimo tempo, il che conferma l'efficacia delle strategie di potatura implementate e dell'euristica adottata.

2.2.4 Istanza 2

Il piano generato è il seguente:

```
* Starting ASTAR search with AJUSTED_SUM heuristic and EslHeuristic By Carmelo-Carmelo-Vittorio
* ASTAR search succeeded
```

found plan as follows:

```
00: (          fill-box r1 b2 food depot) [0]
01: (          fill-box r1 b1 tool depot) [0]
02: (    load-carrier r1 b3 cb s2-cb depot) [0]
```

```

03: (          hold-carrier r1 cb depot) [0]
04: (          hold-carrier r2 ca depot) [0]
05: (    load-carrier r1 b1 ca s1-ca depot) [0]
06: (          move-carrier r2 depot l4 ca) [0]
07: (          give-content r2 p5 tool b1 l4) [0]
08: (          move-carrier r2 l4 depot ca) [0]
09: (          fill-box r1 b1 medicine depot) [0]
10: (          move-carrier r2 depot l1 ca) [0]
11: (    give-content r2 p2 medicine b1 l1) [0]
12: (          move-carrier r2 l1 depot ca) [0]
13: (          fill-box r1 b1 medicine depot) [0]
14: (          move-carrier r2 depot l2 ca) [0]
15: (    give-content r2 p3 medicine b1 l2) [0]
16: (          move-carrier r2 l2 depot ca) [0]
17: (          fill-box r1 b3 tool depot) [0]
18: (          move-carrier r1 depot l5 cb) [0]
19: (          give-content r1 p6 tool b3 l5) [0]
20: (    load-carrier r2 b2 ca s2-ca depot) [0]
21: (          fill-box r2 b1 medicine depot) [0]
22: (          move-carrier r2 depot l3 ca) [0]
23: (          move-carrier r1 l5 depot cb) [0]
24: (          give-content r2 p4 food b2 l3) [0]
25: (    give-content r2 p4 medicine b1 l3) [0]
26: (          move-carrier r2 l3 depot ca) [0]
27: (          fill-box r1 b1 food depot) [0]
28: (          fill-box r1 b2 medicine depot) [0]
29: (          move-carrier r2 depot l4 ca) [0]
30: (          give-content r2 p5 food b1 l4) [0]
31: (    give-content r2 p5 medicine b2 l4) [0]
32: (          fill-box r1 b3 food depot) [0]
33: (          move-carrier r1 depot l5 cb) [0]
34: (          give-content r1 p6 food b3 l5) [0]
35: (          move-carrier r2 l4 depot ca) [0]
36: (          fill-box r2 b1 medicine depot) [0]
37: (          fill-box r2 b2 tool depot) [0]
38: (          move-carrier r2 depot l5 ca) [0]
39: (    give-content r1 p6 medicine b1 l5) [0]
40: (          move-carrier r2 l5 l1 ca) [0]
41: (          give-content r2 p1 tool b2 l1) [0]
42: (satisfied-with-at-least-one p1 food tool) [0]

```

```

time spent:      0,03 seconds parsing
                0,16 seconds encoding
                0,46 seconds searching
                0,65 seconds total time

```

```

memory used:     5,08 MBytes for problem representation
                0,32 MBytes for searching
                5,40 MBytes total

```

Anche la seconda istanza è stata risolta rapidamente, evidenziando l'efficienza dell'algoritmo di pianificazione. Inoltre, è interessante notare che le azioni compiute dai due agenti, r1 e r2, risultano bilanciate, con 17 azioni eseguite da r1 e 25 da r2. Ciò suggerisce una distribuzione equa delle attività tra gli agenti coinvolti nell'esecuzione del piano.

2.2.5 Istanza 3

Il piano generato è il seguente:

* Starting ASTAR search with AJUSTED_SUM heuristic and EslHeuristic By Carmelo-Carmelo-Vittorio

* ASTAR search succeeded

found plan as follows:

```
00: (          fill-box r1 b3 tool depot) [0]
01: (          fill-box r1 b4 food depot) [0]
02: (    load-carrier r1 b4 cb s2-cb depot) [0]
03: (          hold-carrier r1 cb depot) [0]
04: (    load-carrier r1 b1 cb s1-cb depot) [0]
05: (          hold-carrier r2 ca depot) [0]
06: (    load-carrier r1 b2 ca s1-ca depot) [0]
07: (          fill-box r1 b1 medicine depot) [0]
08: (          move-carrier r1 depot l1 cb) [0]
09: (    give-content r1 p2 medicine b1 l1) [0]
10: (    give-content r1 p1 food b4 l1) [0]
11: (satisfied-with-at-least-one p1 food tool) [0]
12: (          move-carrier r1 l1 depot cb) [0]
13: (          fill-box r1 b1 food depot) [0]
14: (          fill-box r1 b4 medicine depot) [0]
15: (          move-carrier r1 depot l3 cb) [0]
16: (    give-content r1 p4 food b1 l3) [0]
17: (    give-content r1 p4 medicine b4 l3) [0]
18: (          move-carrier r1 l3 depot cb) [0]
19: (          fill-box r1 b1 food depot) [0]
20: (          fill-box r1 b4 medicine depot) [0]
21: (          move-carrier r1 depot l4 cb) [0]
22: (    give-content r1 p5 food b1 l4) [0]
23: (    give-content r1 p5 medicine b4 l4) [0]
24: (          move-carrier r1 l4 depot cb) [0]
25: (          fill-box r1 b1 food depot) [0]
26: (          fill-box r1 b4 medicine depot) [0]
27: (          move-carrier r1 depot l5 cb) [0]
28: (    give-content r1 p6 food b1 l5) [0]
29: (    give-content r1 p6 medicine b4 l5) [0]
30: (          move-carrier r1 l5 depot cb) [0]
31: (          fill-box r1 b1 food depot) [0]
32: (          fill-box r1 b4 medicine depot) [0]
33: (          move-carrier r1 depot l6 cb) [0]
34: (    give-content r1 p7 food b1 l6) [0]
35: (    give-content r1 p7 medicine b4 l6) [0]
36: (          move-carrier r1 l6 depot cb) [0]
37: (          fill-box r1 b1 food depot) [0]
38: (          fill-box r1 b4 medicine depot) [0]
39: (          move-carrier r1 depot l7 cb) [0]
40: (    give-content r1 p8 food b1 l7) [0]
41: (    give-content r1 p8 medicine b4 l7) [0]
42: (    load-carrier r2 b3 ca s2-ca depot) [0]
43: (          fill-box r2 b2 medicine depot) [0]
44: (          move-carrier r2 depot l2 ca) [0]
45: (    give-content r2 p3 medicine b2 l2) [0]
46: (          move-carrier r2 l2 l4 ca) [0]
47: (    give-content r2 p5 tool b3 l4) [0]
48: (          move-carrier r1 l7 depot cb) [0]
49: (          fill-box r1 b1 tool depot) [0]
50: (          move-carrier r1 depot l5 cb) [0]
51: (    give-content r1 p6 tool b1 l5) [0]
52: (          move-carrier r1 l5 depot cb) [0]
53: (          fill-box r1 b1 tool depot) [0]
54: (          move-carrier r1 depot l6 cb) [0]
55: (    give-content r1 p7 tool b1 l6) [0]
56: (          move-carrier r1 l6 depot cb) [0]
```

```

57: (          fill-box r1 b1 tool depot) [0]
58: (          move-carrier r1 depot 17 cb) [0]
59: (          give-content r1 p8 tool b1 17) [0]

```

```

time spent:    0,03 seconds parsing
               0,19 seconds encoding
               0,76 seconds searching
               0,98 seconds total time

```

```

memory used:   9,11 MBytes for problem representation
               0,50 MBytes for searching
               9,61 MBytes total

```

Nella terza istanza, il planner ha nuovamente dimostrato un'efficacia notevole risolvendo il problema in poco tempo. Tuttavia, si nota una differenza rispetto alle istanze precedenti, con un numero di azioni sbilanciato tra r1 e r2. In particolare, r1 ha eseguito più di 30 azioni, mentre r2 ne ha eseguite circa una decina.

2.2.6 Riepilogo

Istanza	Risultati		
	Tempo Totale (s)	Azioni Totali	Memoria Utilizzata (MBytes)
1	0.27	14	1.34
2	0.65	42	5.40
3	0.92	59	9.61

Tabella 2.1: Risultati delle istanze

Capitolo 3

Temporal Planning & Robotics

In questa sezione sono state apportate delle modifiche al dominio discusso nel primo capitolo, introducendo una componente temporale al suo interno per renderlo eseguibile dai planner di planUtils e planSys2. In particolare si è deciso di attribuire una durata specifica a ciascuna azione compiuta dall'agente, tenendo conto della tipologia degli oggetti coinvolti, che ora hanno un peso diverso:

- Le medicine (medicine) hanno peso pari ad 1;
- il cibo ha peso pari a 2;
- gli strumenti (tools) hanno peso pari a 3.

Quindi la presenza di oggetti con peso maggiore, per ogni azione, richiederà una tempistica più estesa per garantire una gestione efficace delle richieste.

3.1 Modifica del dominio

Il passo iniziale è stato quello di modificare nel file di dominio la sezione “requirements”, mediante l'aggiunta della libreria “durative-actions”, che ha permesso di aggiungere la componente temporale in ciascuna azione:

```
(:requirements
  :strips :typing
  :equality
  :existential-preconditions
  :numeric-fluents
  :durative-actions :adl
)
```

I tipi precedentemente definiti sono rimasti inalterati, poiché la rappresentazione dell'ambiente in cui l'agente svolgerà le proprie attività è la stessa.

Per quanto concerne la definizione dei diversi pesi sono state utilizzate le functions, che in linea generale assegnano valori a determinati tipi di argomenti. In particolare:

```
(:functions
  (content-weight ?elem - content)
  (box-weight ?b - box)
  (carrier-weight ?c - carrier)
)
```

- (*content-weight ? elem - content*) : funzione che restituisce il peso associato ad un elemento di tipo content;
- (*box-weight ?b - box*): funzione che restituisce il peso associato ad una box;
- (*carrier-weight ?c - carrier*): funzione che restituisce il peso associato ad un carrello.

La modifica più significativa consiste nella sostituzione delle “action” di base, con delle “durative-action”. Si passa quindi da una definizione classica comprendente parametri, precondizioni ed effetti, ad una definizione costituita da parametri, duration, condizioni ed effetti. Duration, come dice la parola stessa, indica la durata di ciascuna

azione. Tale durata può essere specificata come un valore fisso oppure come una espressione matematica variabile, in base agli oggetti coinvolti. Nel contesto delle durative-action, le pre-condizioni tradizionali da valutare prima di eseguire l'azione risultano insufficienti. Pertanto, sono state rimpiazzate da condizioni più elaborate, espresse attraverso i seguenti costrutti:

- **At start:** indica che una determinata condizione deve essere verificata prima dell'avvio dell'azione stessa;
- **over all:** indica che una condizione deve essere verificata prima dell'inizio dell'azione e deve mantenersi valida per l'intera durata, fino al suo completamento.

Allo stesso modo, poichè possono verificarsi conseguenze sulla base di conoscenza sia all'inizio che alla fine dell'azione, gli effetti vengono descritti utilizzando i seguenti costrutti:

- **At start:** indica che una condizione sarà valida appena l'azione inizia;
- **at end:** indica che una condizione sarà valida appena l'azione si conclude.

Inoltre sono stati introdotti i seguenti predicati:

```
(is-taken ?c - carrier); carrier ?c it is not available
(hands-free ?r - robot) ; robot ?r is not holding a carrier
(idle ?r - robot) ; robot ?r is idle
```

- *(idle ?r - robot)*: Il seguente predicato è stato introdotto per garantire la mutua-esclusività delle durative-action. Inizialmente, tutti i robot di ciascuna istanza sono contrassegnati come “idle” per definire il loro stato inattivo all'avvio. Pertanto, in ogni azione che verrà eseguita dai robot, si ha come condizione iniziale il loro stato di inattività. Una volta verificata, tale condizione porta ad una conseguenza, posta all'interno della clausola effect, che specifica il passaggio allo stato attivo del robot che ha intrapreso l'azione. Al termine dell'azione, si specifica che il robot deve ritornare nuovamente allo stato di inattività, pronto per intraprendere una nuova azione. Questo approccio assicura che, in qualsiasi momento, un robot sia impegnato in una sola azione, garantendo la corretta sequenza e la mutua esclusività delle attività.
- *(is-taken ?c -carrier)* e *(hands-free ?r - robot)* : Entrambi i predicati sono stati introdotti poiché i planner temporali, come lpg-td, popf o altri, non supportano i costrutti “forall” ed “exist”.

La mancanza del forall, all'interno dell'azione durativa “move-carrier”, non ci ha permesso di tenere traccia della posizione delle box. Per risolvere tale problematica abbiamo supposto che la posizione delle casse sia la stessa del carrello su cui sono caricate. La mancanza del costrutto exist, invece, all'interno dell'azione durativa “move”, è stata risolta con l'aggiunta del predicato “hands-free”.

L'azione “move”, come detto nel primo capitolo, è stata concepita per consentire al robot di spostarsi da una posizione all'altra; l'exist è servito per assicurarci che il robot non stesse portando con sé alcun carrello, come vediamo di seguito:

Listing 3.1: Modifiche azione MOVE

```
; * PRIMA
(not
  (exists (?x - carrier)
    (is-holding ?r ?x)
  )
)
; * DOPO
(over all
  (hands-free ?r)
)
```

“Is-taken” è stato utilizzato, in sostituzione dell'exist, all'interno della durative-action “hold-carrier” per garantire che nessun altro robot sia già in possesso del carrello desiderato.


```

; * PRIMA
(not
  (exists (?x - robot)
    (is-holding ?x ?c)
  )
)
; * DOPO
(at start (and (idle ?r) (not (is-taken ?c))))

```

Di seguito viene riportata per semplicità soltanto una durative-actions, che però ci mostra alcune importanti differenze rispetto al vecchio dominio.

```

(:durative-action fill-box
:parameters ( ?r - robot ?b - box ?elem - content ?l - location)
:duration (= ?duration (content-weight ?elem))
:condition (and
  (at start
    (and
      (not (full ?b))
      (idle ?r)
    )
  )
  (over all
    (and
      (depot-at ?l) (at ?r ?l) (at ?elem ?l)
      (at ?b ?l)
    )
  )
)
:effect (and
  (at start
    (and
      (full ?b)
      (not (idle ?r))
    )
  )
  (at end (increase (box-weight ?b) (content-weight ?elem)))
  (at end
    (and
      (idle ?r)
      (has-inside ?b ?elem)
    )
  )
)
)

```

Come detto nel primo capitolo, l'operazione di fill-box è stata creata per dare la possibilità al robot di riempire una box con un certo contenuto. In questo caso specifico, è stato opportuno considerare la durata dell'azione dipendente esclusivamente dal peso dell'elemento che verrà inserito all'interno della box. Nella clausola effect è possibile notare l'utilizzo delle functions precedentemente introdotte. In particolare, alla fine dell'azione viene eseguito il comando :

$$(increase(box - weight?b)(content - weight?elem))$$

Tale comando incrementa la box con il peso dello specifico elemento che si vuole inserire al suo interno.

3.2 Modifica delle istanze

E' stata modificata di conseguenza l'istanza 1 del problema, come ci è stato richiesto, in modo che sia compatibile con la nuova definizione. Di seguito è riportata la modifica dell'istanza 1 del problema:

```

(define (problem esl-p1-t) (:domain emergency-services-logistics-temporal)

(:objects
  b1 b2 b3 b4 b5 - box
  depot l1 l2 - location
  p1 p2 p3 - person
  r1 - robot
  ca - carrier
  s1-ca s2-ca s3-ca s4-ca - slot
  food medicine tool - content
)

(:init
  (depot-at depot)
  (at b1 depot) (at b2 depot) (at b3 depot) (at b4 depot) (at b5 depot)
  (at food depot) (at medicine depot) (at tool depot)
  (at p1 l1) (at p2 l1) (at p3 l2)
  (at r1 depot) (at ca depot)
  (idle r1)

  (empty s1-ca ca) (empty s2-ca ca)
  (empty s3-ca ca) (empty s4-ca ca)

  ( = (content-weight medicine) 1)
  ( = (content-weight food ) 2)
  ( = (content-weight tool) 3)

  ( = (carrier-weight ca) 0)
  ( = (box-weight b1) 0)
  ( = (box-weight b2) 0)
  ( = (box-weight b3) 0)
  ( = (box-weight b4) 0)
  ( = (box-weight b5) 0)
)

(:goal (and
  (has-content p1 food )
  (has-content p1 medicine)
  (has-content p2 medicine)
  (has-content p3 food)
))

;un-comment the following line if metric is needed
;(:metric minimize (???))
)

```

Gli oggetti e il goal della istanza rimangono perfettamente identici alla vecchia versione. E' stato aggiunto il predicato "idle r1", come anticipato, per indicare lo stato di inattività del robot 1. Di fondamentale importanza sono l'aggiunta delle functions, che specificano i pesi degli oggetti in modo da rispettare i vincoli del problema, inoltre impostano le box e il carrello con un peso pari a 0 poiché si suppone siano vuoti all'inizio.

3.3 Plan Utils

Le modifiche apportate al dominio e alla istanza 1 hanno inserito il nostro problema nella categoria del Temporal Planning. Una libreria che permette di risolvere problemi di questa tipologia è Plan Utils. Per il nostro problema specifico, tra i diversi planner temporali, il più efficiente è risultato **lpg-td** che una volta terminata l'esecuzione ha restituito il seguente output:

```

0.00030:    (FILL-BOX R1 B1 MEDICINE DEPOT) [1.00000]
1.00050:    (HOLD-CARRIER R1 CA DEPOT) [1.00000]
2.00080:    (LOAD-CARRIER R1 B1 CA S4-CA DEPOT) [2.00000]
4.00100:    (FILL-BOX R1 B3 MEDICINE DEPOT) [1.00000]
5.00120:    (LOAD-CARRIER R1 B3 CA S3-CA DEPOT) [2.00000]

```

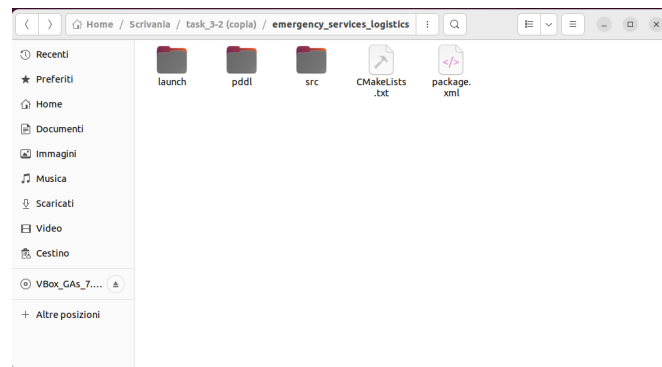
```

7.00150: (FILL-BOX R1 B4 FOOD DEPOT) [2.00000]
9.00170: (LOAD-CARRIER R1 B4 CA S1-CA DEPOT) [2.00000]
11.00200: (FILL-BOX R1 B2 FOOD DEPOT) [2.00000]
13.00220: (LOAD-CARRIER R1 B2 CA S2-CA DEPOT) [2.00000]
15.00250: (MOVE-CARRIER R1 DEPOT L1 CA) [12.00000]
27.00280: (GIVE-CONTENT R1 P1 B1 MEDICINE L1 CA) [2.00000]
29.00300: (GIVE-CONTENT R1 P2 B3 MEDICINE L1 CA) [2.00000]
31.00330: (GIVE-CONTENT R1 P1 B4 FOOD L1 CA) [2.00000]
33.00350: (MOVE-CARRIER R1 L1 L2 CA) [4.00000]
37.00370: (GIVE-CONTENT R1 P3 B2 FOOD L2 CA) [2.00000]

```

3.4 PlanSys2

L'ultima sezione è caratterizzata dall'utilizzo della libreria ROS2 Planning System (PlanSys2), un sistema di pianificazione basato su PDDL, che ci ha permesso di simulare in tempo reale il piano generato al punto precedente. Il primo passo è stato quello di creare il package di PlanSys2 rinominato *"emergency_services_logistics"*.



- **Cartella launch:** La cartella launch contiene due file: *"commands"* e *"plansys2.esl.py"*; nel primo sono contenuti i comandi necessari ad impostare l'istanza del problema nel terminale di ros2. Nel file python abbiamo inserito le istruzioni che ci consentono di avviare Plansys2, selezionare il dominio e lanciare gli eseguibili definiti nel file *"CMakeLists.txt"*.
- **Cartella pddl:** Nella cartella pddl troviamo il file di dominio, opportunamente modificato per supportare le durative actions, *"esl-domain.pddl"* e il file *"esl-p1-plan.plan"* contenente il piano generato dal planner *"lpg-td"* applicato all'istanza uno.
- **Cartella src:** Nella cartella *"src"* sono collocati i sorgenti in linguaggio C++ che ci consentono di definire un nodo per ogni azione definita nel file di dominio. Il metodo principale di ogni file è il *do_work()*, il quale verifica se l'azione è terminata o meno. Di seguito è riportato il codice C++ dell'azione *give-content*.

```

private:
void do_work()
{
    if (progress_ < 1.0) {
        progress_ += 0.2;
        send_feedback(progress_, "Giving content");
    } else {
        finish(true, 1.0, "Content gived");
        progress_ = 0.0;
        std::cout << std::endl;
    }
    std::cout << "\r\e[K" << std::flush;
    std::cout << "Giving content ... [" << std::min(100.0, progress_ * 100.0)
        << "%] " << std::flush;
}

float progress_;
};

```

- **CMakeLists.txt** e **package.xml**: Il file “CMakeLists.txt” ci consente di ricavare dei file eseguibili a partire dai sorgenti C++ collocati nella cartella “src”. Infine, nel file “package.xml” sono contenute informazioni sulla struttura del nostro package

3.4.1 Esecuzione istanza su PlanSys2

L'esecuzione si compone di diversi passaggi:

- Primo Terminale:

1. Compilare i file C++ presenti nella cartella *src*

```
colcon build --symlink-install
```

2. Eseguire il comando

```
source install/setup.bash
```

3. Lanciare il comando

```
ros2 launch emergency_services_logistics plansys2_esl.py
```

- Secondo Terminale:

1. Avviare il terminale di PlanSys2 con il comando:

```
ros2 run plansys2_terminal plansys2_terminal
```

2. Dare in input il file *commands* tramite il comando

```
source ./launch/commands
```

3. Eseguire il piano tramite il comando

```
run plan-file esl-p1-plan.plan
```

Una volta effettuati i seguenti passaggi, all'interno del primo terminale verrà mostrata l'esecuzione passo passo delle azioni, invece sul secondo terminale verrà mostrato il piano completo.

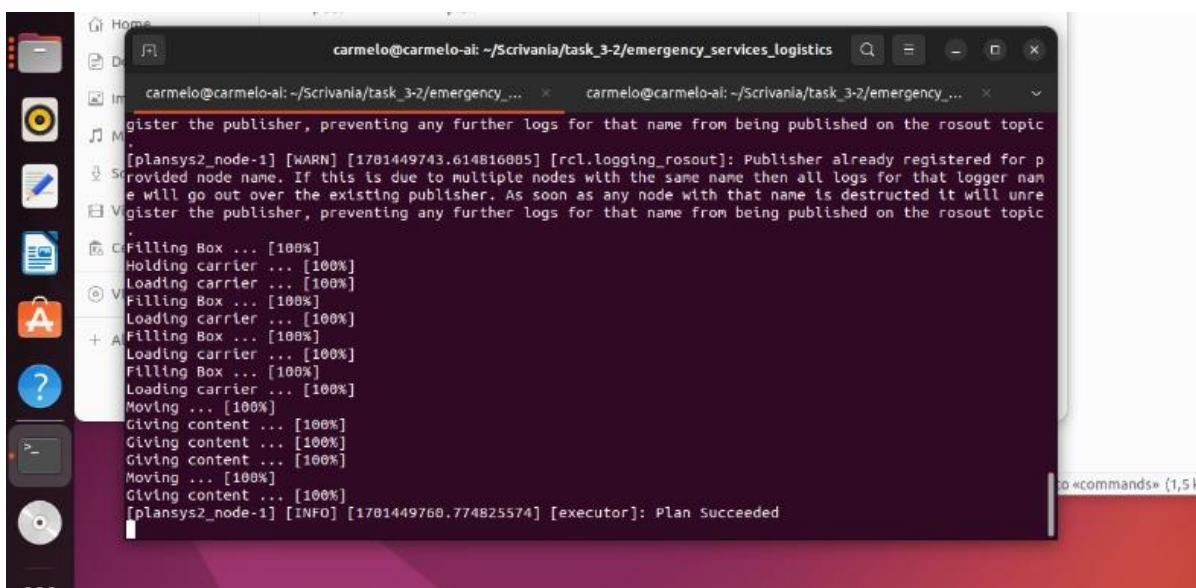
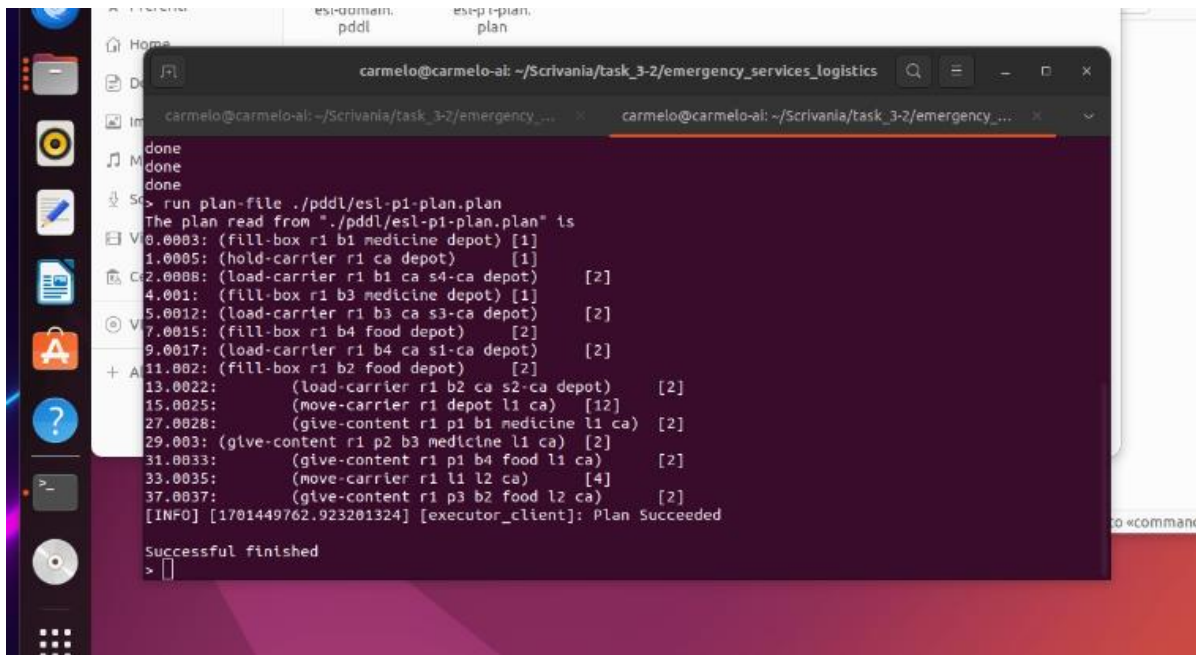


Figura 3.1: Primo Terminale



The screenshot shows a terminal window titled 'carmelo@carmelo-at: ~/Scrivanla/task_3-2/emergency_services_logistics'. The terminal displays the output of a command to run a plan file. The output shows a sequence of actions with timestamps and durations, such as 'fill-box', 'hold-carrier', 'load-carrier', and 'give-content'. The plan concludes with a success message: '[INFO] [1701449762.923201324] [executor_client]: Plan Succeeded' and 'Successful finished'.

```
carmelo@carmelo-at: ~/Scrivanla/task_3-2/emergency_services_logistics
> run plan-file ./pddl/esl-p1-plan.plan
The plan read from "./pddl/esl-p1-plan.plan" is
0.0003: (fill-box r1 b1 medicine depot) [1]
1.0005: (hold-carrier r1 ca depot) [1]
2.0008: (load-carrier r1 b1 ca s4-ca depot) [2]
4.001: (fill-box r1 b3 medicine depot) [1]
5.0012: (load-carrier r1 b3 ca s3-ca depot) [2]
7.0015: (fill-box r1 b4 food depot) [2]
9.0017: (load-carrier r1 b4 ca s1-ca depot) [2]
11.002: (fill-box r1 b2 food depot) [2]
13.0022: (load-carrier r1 b2 ca s2-ca depot) [2]
15.0025: (move-carrier r1 depot l1 ca) [12]
27.0028: (give-content r1 p1 b1 medicine l1 ca) [2]
29.003: (give-content r1 p2 b3 medicine l1 ca) [2]
31.0033: (give-content r1 p1 b4 food l1 ca) [2]
33.0035: (move-carrier r1 l1 l2 ca) [4]
37.0037: (give-content r1 p3 b2 food l2 ca) [2]
[INFO] [1701449762.923201324] [executor_client]: Plan Succeeded

Successful finished
>
```

Figura 3.2: Secondo Terminale

Capitolo 4

Organizzazione dell'archivio consegnato

L'archivio è così organizzato:

- **Cartella task_1:** All'interno della cartella è presente il file PDDL del dominio sviluppato seguendo i punti richiesti dalla traccia (*esl-domain.pddl*)
- **Cartella task_2:** In questa cartella è presente uno script *run.sh* che permette di avviare la risoluzione dei problemi (*./run.sh <numero istanza da risolvere>*). Inoltre sono presenti le seguenti sotto cartelle:
 - *src*: contiene i file Java riguardanti l'implementazione dell'euristica e degli algoritmi di ricerca, come già discusso nel Capitolo 2.
 - *lib*: contiene la libreria PDDL4J in formato .jar.
 - *pddl_instances*: contiene i file, in formato .pddl, riguardanti la modellazione del dominio e delle istanze richieste.
- **Cartella task_3_1:** Contiene i file PDDL del dominio e dell'istanza 1 opportunamente modificati per il Temporal Planning. Inoltre è presente il piano generato da lpg-td (*esl-p1-plan.txt*).
- **Cartella task_3_2:** L'organizzazione della cartella è identica a come spiegato in 3.4