Roblocks: Risoluzione Automatica di Problemi di Spostamento Blocchi con ASP

Daniele Molinari
Università degli studi di Parma, Italia
daniele.molinari3@studenti.unipr.it

Francesco Bernini
Università degli studi di Parma, Italia
francesco.bernini1@studenti.unipr.it

ABSTRACT

La ricerca delle mosse ottimali in un problema di pianificazione, come quello affrontato in questo progetto riguardante lo spostamento e la disposizione di blocchi in una griglia, può risultare estremamente complessa. L'automazione della ricerca efficiente delle mosse ottimali è fondamentale per gestire l'elevato numero di scenari possibili, necessari per ottenere il risultato desiderato. In questo contesto, è stato sviluppato un sistema basato su **Answer Set Programming** (ASP), un paradigma logico particolarmente adatto alla risoluzione di questa tipologia di problemi. Utilizzando script bash, Python e clingo, un solver per ASP, sono state implementate soluzioni in grado di identificare e visualizzare graficamente le mosse necessarie per raggiungere una disposizione ottimale dei blocchi.

1 INTRODUZIONE

L'obiettivo principale del progetto è stato sviluppare un sistema per la risoluzione del problema di pianificazione dello spostamento di blocchi all'interno di una griglia. Le specifiche sono state implementate e successivamente verificate attraverso il solver clingo, utilizzando una serie di benchmark per testarne l'efficacia e convalidarne il corretto funzionamento.

1.1 Background

Il paradigma di programmazione dichiarativa adottato è l'**Answer Set Programming** (ASP), particolarmente efficace nella risoluzione di problemi di ricerca complessi, come quelli appartenenti alla classe *NP-completi*. Con esso, la ricerca delle soluzioni si riduce all'identificazione di modelli stabili, ottenuti tramite l'uso di *Answer Set Solver*, strumenti progettati per generarli e individuarli in modo efficiente.

La principale differenza tra la programmazione dichiarativa, come quella offerta da ASP, e la programmazione imperativa risiede nell'approccio alla risoluzione dei problemi: nella programmazione dichiarativa, ci si concentra su **cosa** deve essere ottenuto, anziché **come** raggiungere il risultato. Questa filosofia è il principio fondante della programmazione dichiarativa e ne definisce il contrasto con l'approccio imperativo.

In questo contesto, per l'identificazione dei modelli stabili nel problema affrontato è stato utilizzato **Clingo**, un *solver* open-source sviluppato dal gruppo di ricerca Potassco.

Clingo combina in un unico strumento due componenti fondamentali per l'Answer Set Programming (ASP):

- **Gringo**: responsabile della fase di *grounding*, ovvero della trasformazione del programma dichiarativo eliminando le variabili e generando una rappresentazione puramente proposizionale.
- **Clasp**: responsabile della fase di *solving*, ovvero di trovare i modelli stabili attraverso tecniche avanzate di ricerca combinatoria.

La sintassi di Clingo consente di definire *regole*, *vincoli* e *ottimizzazioni*, elementi cruciali per migliorare l'efficienza delle fasi di *grounding* e *solving*. Un altro aspetto significativo sfruttato nel progetto è il **Bash scripting**, utilizzato per semplificare e automatizzare la gestione delle invocazioni del solver e la manipolazione delle soluzioni ottenute, rendendo l'intero processo più efficiente e flessibile.

1.2 Problema

Il problema considerato riguarda la pianificazione degli spostamenti di un insieme di blocchi all'interno di una griglia di dimensioni $m \times n$. Ogni blocco b_i è di forma quadrata con lato di dimensione intera s_i ed è inizialmente posizionato in una cella specifica della griglia, nota attraverso le coordinate del suo vertice inferiore sinistro.
I blocchi possono essere spostati solo utilizzando l'operazione di move(b_i, D, t), dove D rappresenta la direzione dello spostamento, che può essere verso nord (N), sud (S), ovest (W) o est (E) al tempo t. Tuttavia, il movimento è soggetto a vincoli:

- Gli spostamenti sono unitari e consentono solo la **spinta**, ovvero un blocco può essere spostato solo nella direzione in cui si trova lo spazio libero, ma non può essere tirato.
- Se un blocco raggiunge il bordo sinistro o destro della griglia, non potrà più essere spostato orizzontalmente. In modo analogo per quanto riguarda il bordo in alto o in basso per la direzione verticale.
- Una mossa è valida solo se la posizione di destinazione è libera, ovvero non occupata da un altro blocco.

L'obiettivo del problema è quello di **ammassare i blocchi** nella parte inferiore della griglia, a partire dall'angolo in basso a sinistra. Se lo spazio alla base non è sufficiente, i blocchi devono essere impilati, cercando di minimizzare l'altezza massima raggiunta.

Il problema viene affrontato in due fasi principali:

(1) **Ottimizzazione statica**: determinare una sistemazione ottimale dei blocchi che minimizzi l'altezza totale, senza considerare le restrizioni imposte dagli spostamenti consentiti. Questo caso si può vedere in Figura 3.
(2) **Pianificazione dei movimenti**: trovare una sequenza di mosse che consenta di raggiungere la configurazione ottimale identificata nella fase precedente, rispettando i vincoli di movimento.

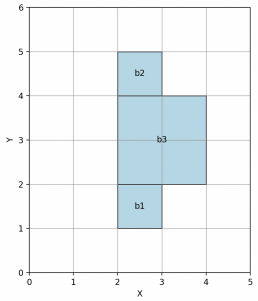

Figure 1: Situazione iniziale

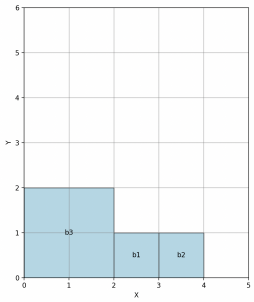

Figure 2: Situazione finale

Figure 3: Un modello data una situazione iniziale

2 IMPLEMENTAZIONE

In questa sezione verranno descritte le scelte implementative prese, sia quelle relative all'attuazione delle specifiche, sia quelle relative all'ottimizzazione dei tempi di calcolo per entrambe le fasi del problema.

2.1 Ottimizzazione statica

Per quanto riguarda la prima parte (implementata nel file `init.asp`) sono state definite regole e vincoli basate sulla configurazione iniziale fornita nel seguente modo:

```
#const max_width=5.
#const max_height=6.
init_block(b1,1,2,1).
init_block(b2,1,2,4).
init_block(b3,2,2,2).
```

dove le costanti `max_width` e `max_height` definiscono la dimensione della griglia e `init_block(ID,DIM,X,Y)` sono i predicati che definiscono le informazioni essenziali sul posizionamento dei blocchi.

Con i seguenti vincoli vengono impedite situazioni iniziali in input con numero e/o dimensioni di blocchi troppo elevate per la griglia (durante la ricerca della configurazione ottima, la raggiungibilità dalla posizione iniziale non viene considerata):

```
% Un blocco non può sovrapporsi ad un altro
:- init_block(ID1,DIM1,X1,Y1),
   init_block(ID2,DIM2,X2,Y2),
   ID1 != ID2,
   X1 < X2+DIM2, X1 > X2-1,
   Y1 < Y2+DIM2, Y1 > Y2-1.

% Un blocco non può uscire dalla griglia
:- init_block(ID,DIM,X,Y),
   (X + DIM - 1) > (max_width-1).

:- init_block(ID,DIM,X,Y),
   (Y + DIM - 1) > (max_height-1).
```

Tramite intervalli vengono definite tutte le possibili coordinate assegnabili:

```
width(0..max_width-1).
height(0..max_height-1).
```

Una volta definita la situazione iniziale, vengono individuate le configurazioni migliori secondo le specifiche. Come prima cosa, vengono generate tutte le posizioni che un blocco in posizione finale può assumere (banale combinazione di tutte le coordinate possibili):

```
1 { goal_block(ID,DIM,X,Y) : width(X), height(Y)} 1 :-
↪  init_block(ID,DIM,_,_).
```

Poi, con vincoli identici a quelli per i blocchi iniziali, vengono eliminate le sovrapposizioni e le posizioni che porterebbero parti del blocco a fuoriuscire dalla griglia.

Vengono definiti poi 2 predicati che permettono l'eliminazione di altre configurazioni non ottimali.

Il primo è `supported(ID)` che indica quali blocchi hanno direttamente sotto di essi altri blocchi di uguale o maggiore dimensione:

```
supported(ID1) :-
    goal_block(ID1,DIM1,X1,Y1),
    goal_block(ID2,DIM2,X2,Y2),
    ID1 != ID2,
    Y1 = Y2 + DIM2,
    X1 >= X2,
    X1+DIM1-1 < X2 + DIM2.
```

Esso è utile per eliminare quelle configurazioni che prevedono blocchi grandi sopra blocchi piccoli escludendo però quei blocchi che si trovano sul "fondo" perché non potrebbero avere alcun blocco al di sotto di essi:

```
:- goal_block(ID1,DIM1,X1,Y1),
   Y1 > 0,
   not supported(ID1).
```

Il secondo invece è `occupied_left(DIM1,X,Y)` che indica quali blocchi hanno la propria sinistra occupata da un blocco uguale o più grande:

```
occupied_left(DIM1,X,Y) :-
    width(X),
    height(Y),
    goal_block(_,DIM1,X,Y),
    goal_block(_,DIM2,X2,Y2),
    DIM1 <= DIM2,
    X2 + DIM2 = X,
    Y >= Y2,
    Y < Y2 + DIM2.
```

Con questo è possibile eliminare quelle configurazioni che prevedono blocchi piccoli alla sinistra di blocchi grandi escludendo però

dal controllo quei blocchi che si trovano sul lato sinistro della griglia e non potrebbero avere alcun blocco alla propria sinistra:

```
:- goal_block(ID1,DIM1,X1,Y1),
   X1 > 0,
   not occupied_left(DIM1,X1,Y1).
```

Con le sole righe di codice mostrate precedentemente, le configurazioni possibili ammetterebbero "agglomerati" di blocchi a qualsiasi altitudine. Per minimizzare l'altezza dei blocchi impilati vengono utilizzati due aggregati di ottimizzazione:

```
#minimize {Y+DIM-1,Y: goal_block(_,DIM,X,Y)}.

#minimize { Y * DIM : goal_block(_,DIM,_,Y) }.
```

Il primo preferisce quelle configurazioni che hanno altezza più bassa e il secondo preferisce quelle che hanno blocchi più grandi in basso.

La combinazioni di queste regole e vincoli producono più configurazioni ottime (multiple serie di `goal_block`) tra le quali una sola verrà fornita al secondo programma (`main.asp`) tramite uno script insieme alla situazione iniziale (`init_block`) e alle dimensioni della griglia.

E' importante notare come alcune configurazioni ottimali ma superflue vengano escluse dalla ricerca (come quella in Figura 4) per ridurre i tempi di ricerca.

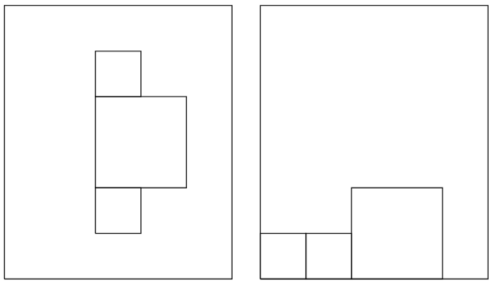

Figure 4: Esempio di stato finale (a destra) considerato superfluo

2.2 Pianificazione dei movimenti

L'implementazione della seconda parte è stata affrontata con la modalità incrementale offerta da `Clingo` che permette di cercare la soluzione al problema più volte, sostituendo ad un atomo (`t` in questo caso) un valore incrementato ad ogni iterazione, sfruttato come tempo discreto, con l'enorme vantaggio garantito dalla conservazione delle deduzioni già calcolate all'iterazione precedente per quella successiva.

Nella sezione che include le informazioni uguali per ogni iterazione sono state definite tutte le coordinate possibili come in `init.asp`, le quattro direzioni possibili di mossa con il loro delta da applicare alle coordinate dei blocchi per determinarne lo spostamento e alcuni predicati autoesplicativi che sono risultati comodi durante l'implementazione:

```
#program base.

% Posizioni possibili
width(0..max_width-1).
height(0..max_height-1).

% Direzioni di movimento
direction(n,0,1).
direction(s,0,-1).
direction(e,1,0).
direction(w,-1,0).

% Direzioni opposte
opposite(e,w).
opposite(w,e).
opposite(n,s).
opposite(s,n).

% Bordi della griglia
borderX(0).
borderX(max_width).
borderY(0).
borderY(max_height).
```

Inoltre, vengono effettuati dei controlli preliminari sulla raggiungibilità della situazione di goal a partire da quella iniziale:

```
% Bordi superiore e inferiore
unreachable_target :-
   init_block(ID,DIM,_,Y),
   borderY(Y+DIM;Y),
   not goal_block(ID,DIM,_,Y).

% Bordi destro e sinistro
...
```

Queste due regole rendono vera la costante `unreachable_target` nel caso in cui un blocco si trovasse inizialmente su un bordo diverso da quello che dovrebbe raggiungere. Infatti, il blocco non potrà mai staccarsi dal bordo, rendendo inutile la ricerca di mosse (questa è in tutto e per tutto una condizione di terminazione che verrà controllata anche al tempo 0).

Un altro caso che rende vani i tentativi di ricerca delle mosse è quello dove il posizionamento iniziale impedisce di muovere tutti i blocchi. Per evitarlo, sono state definite le seguenti regole:

```
% Bordo destro e superiore
blocked_block(ID) :- init_block(ID,DIM,X,Y),
   #count { VAL : init_block(ID1, DIM1, X1, Y1),
                  ID!=ID1,
                  X1 + DIM1 = X,
      %* e *%     VAL = Y..(Y + DIM - 1),
                  VAL < Y1 + DIM1,
                  VAL + DIM1 -1 >= Y1
          } >= DIM,
   #count { VAL : init_block(ID3, DIM3, X3, Y3),
                  ID!=ID3,
                  Y3 + DIM3 = Y,
      %* n *%     VAL = X..(X + DIM - 1),
                  VAL < X3 + DIM3,
                  VAL + DIM3 -1 >= X3
          } >= DIM.

% Bordo destro e inferiore
...

% Bordo sinistro e superiore
...

% Bordo sinistro e inferiore
...
```

```
unreachable_target :- #count {ID : blocked_block(ID)   } = Nblocked,
                      #count {ID : init_block(ID,_,_,_)} = Nblocks,
                      Nblocked = Nblocks.
```

Quindi, la condizione di irraggiungibilità è verificata nel caso in cui tutti i blocchi abbiano almeno due 2 bordi non opposti completamente occupati.

Dopo i controlli preliminari segue un'inizializzazione dei predicati che "sostituiscono" `init_block` e `goal_block`:

```
% Posizionamento dei blocchi nella configurazione iniziale (al
↪  tempo 0)
at(DIM,X,Y,0) :- init_block(_,DIM,X,Y).

% Import della configurazione goal
target(DIM,X,Y) :- goal_block(_,DIM,X,Y).
```

La scelta di identificare ogni blocco solamente tramite la propria posizione è dovuta a motivi di performance nella ricerca delle mosse: è più rapido cercare di spostare blocchi in una posizione di goal qualsiasi verificando solo che la dimensione sia corretta piuttosto che cercare le mosse che spostano ciascun blocco nell'unica posizione di goal associata al proprio `ID`.

La fase centrale del programma è quella dove vengono definite le regole e i vincoli da verificare ad ogni `t` (partendo da `t=1`).

Come prima cosa vengono generate tutte le mosse come combinazioni di posizioni dei blocchi al momento precedente (`t-1`) e direzioni possibili:

```
#program step(t).

1 { move(DIM,X,Y,D,t) : direction(D,DX,DY), at(DIM,X,Y,t-1),
↪  width(X), height(Y) } 1.
```

Sulla base di esse vengono calcolate tutte le nuove posizioni dei blocchi al tempo `t` in modo che solo un blocco alla volta possa muoversi nelle 4 direzioni e gli altri rimangano fermi:

```
% Calcolo nuove posizioni
at(DIM, X+DX, Y+DY, t) :-
    at(DIM, X, Y, t-1),
    move(DIM, X, Y, D, t),
    direction(D, DX, DY),
    width(X+DX),
    height(Y+DY).

% Inerzia: Se il blocco non si muove, rimane fermo
at(DIM, X, Y, t) :-
    at(DIM, X, Y, t-1),
    not move(DIM,X,Y,_,t).
```

Solo con queste indicazioni il numero di combinazioni sarebbe elevatissimo, anche per piccole griglie. Perciò, sono stati pensati vincoli che escludano mosse impossibili e mosse inutili.

Sono state quindi eliminate quelle mosse che avrebbero portato un blocco a sovrapporsi ad un altro:

```
:- move(DIM, X, Y, D, t),
   direction(D, DX, DY),
   X_new = X + DX,
   Y_new = Y + DY,
   at(DIM1, X1, Y1, t-1),
   not move(DIM1, X1, Y1, _, t),
   X_new < X1+DIM1, X_new+DIM-1 > X1-1,
   Y_new < Y1+DIM1, Y_new+DIM-1 > Y1-1.
```

E quelle che avrebbero mosso il blocco al di fuori della griglia o lontano dal bordo nonostante l'impossibilità di spinta:

```
:- move(DIM, X, Y, e, t), at(DIM,X,_,t-1), (X + DIM) = max_width.
:- move(DIM, X, Y, w, t), at(DIM,X,_,t-1), (X + DIM) = max_width.
:- move(DIM, X, Y, e, t), at(DIM,X,_,t-1), X = 0.
:- move(DIM, X, Y, w, t), at(DIM,X,_,t-1), X = 0.

... % simile per "n" e "s"
```

Inoltre, un blocco non può essere spinto da un lato se esso risulta completamente occupato da altri blocchi, perciò è necessario eliminare anche quelle mosse:

```
:- move(DIM,X,Y,e,t),
   #count { VAL : VAL = Y..(Y + DIM - 1), at(DIM1, X1, Y1, t-1),
        X1 + DIM1 = X, VAL < Y1 + DIM1, VAL + DIM1 -1 >= Y1} >= DIM.

... % simile per "w", "n" e "s"
```

Per quanto riguarda le mosse inutili invece, sono state eliminate le coppie di mosse consecutive e opposte di uno stesso blocco:

```
:- move(DIM,X+OX,Y+OY,D,t),
   opposite(D,O),
   direction(O,OX,OY),
   move(DIM,X,Y,O,t-1).
```

E quelle mosse che portano un blocco su un bordo diverso da quello che avrebbe dovuto raggiungere (anche se nessuno):

```
:- move(DIM,X,_,D,t),
   direction(D,DX,_),
   borderX(X+DIM+DX),
   not target(DIM,X+DX,_),
   N1 = #count { Y1 : at(DIM1, X1, Y1, t), move(DIM1,_,_,_,t),
   ↪  X1+DIM1 = max_width},
   N2 = #count { Y1 : target(DIM1, X1, Y1), move(DIM1,_,_,_,t),
   ↪  X1+DIM1 = max_width},
   N1 > N2.

... % simile per il bordo sinistro, superiore e inferiore
```

Nella sezione finale, quella che controlla il raggiungimento della situazione di goal al tempo `t`, sono state definite le condizioni di terminazione del programma:

```
#program check(t).

% Verifica se ogni blocco e' al goal
reached_target(DIM, X, Y, t) :-
    at(DIM, X, Y, t),
    target(DIM, X, Y).

% Termina se ogni blocco e' al goal
goal(t) :-
    reached_target(DIM, X, Y, t) : goal_block(_,DIM,X,Y).

% Termina se la configurazione iniziale non permette soluzioni
goal(t) :-
    unreachable_target.

% Innesca la verifica del goal ad ogni t
:- query(t), not goal(t).
```

Ovvero, se al tempo `t` (il cui valore è istanziato ad ogni `step(t)` da `query(t)`) almeno un target risulta non raggiungibile o se tutte le posizioni goal sono state raggiunte da un blocco.

3 BENCHMARK

Per la fase di benchmark, le istruzioni di consegna del progetto suggerivano di creare 100 casi di test con complessità crescente, legata sia alla dimensione che alla quantità dei blocchi. A tale scopo, sono stati affrontati due principali obiettivi:

- **Correttezza**: La correttezza dei dati di benchmark è stata fondamentale per garantire che i casi testati fossero sia verosimili che soddisfacibili, in modo da ottenere risultati significativi e realistici.
- **Scalabilità**: Sono stati eseguiti una serie di test per determinare i valori ottimali sui quali eseguire i benchmark, considerando la relazione tra il tempo di risoluzione e la complessità computazionale dei casi di test.

3.1 Correttezza

Per la generazione di scenari realistici è stato scelto di utilizzare la programmazione ASP, così come adottato per affrontare il problema del progetto. Questo ha permesso di esplorare le possibili configurazioni a partire da:

- dimensione fissata della griglia
- numero e disposizione dei blocchi
- dimensioni ammissibili dei blocchi

Attraverso uno script, che analizzato in seguito, sono state parametrizzate le costanti del sistema, permettendo di generare a ogni esecuzione una configurazione casuale tra quelle possibili. Inoltre, è stata imposta la condizione che nella configurazione iniziale i blocchi non tocchino i bordi, evitando così situazioni di partenza da cui sarebbe impossibile muoversi nella fase di definizione dei movimenti. Di seguito, viene presentato il programma ASP chiamato `generator.asp`, che modella le configurazioni possibili.

```
#const max_width = 4.   % Larghezza massima (X)
#const max_height = 5.  % Altezza massima (Y)
#const max_dim = 3. % Dimensione massima dei blocchi generati

width(0..max_width-1). % Larghezza griglia (X)
height(0..max_height-1). % Altezza griglia (Y)

dim(1..max_dim).

% Predicato posizione finale
1 { init_block(ID,DIM,X,Y) : width(X), height(Y), dim(DIM)} 1 :-
↪  block(ID).

% === Vincoli controllo ===
:- init_block(ID1,DIM1,X1,Y1),
   init_block(ID2,DIM2,X2,Y2),
   ID1 != ID2,
   X1 < X2+DIM2, X1+DIM1-1 > X2-1,
   Y1 < Y2+DIM2, Y1+DIM1-1 > Y2-1.

% Non genero blocchi fuori la griglia o sui bordi
:- init_block(ID,DIM,X,Y),
   (X + DIM - 1) > (max_width-2).

:- init_block(ID,DIM,X,Y),
   (Y + DIM - 1) > (max_height-2).

:- init_block(ID,DIM,X,Y),
   X < 1.

:- init_block(ID,DIM,X,Y),
   Y < 1.

#maximize { DIM : init_block(_,DIM,_,_), dim(DIM) }.

#show init_block/4.
#show block/1.
```

Per eseguire il programma (`generator.asp`), è stato utilizzato il seguente comando, che permette di ottenere una configurazione casuale:

```
clingo tmp_generator.asp generator.asp \
        --rand-freq=1 \
        --seed=$RANDOM \
        --const max_width=$max_width \
        --const max_height=$max_height \
        --const max_dim=$max_dim
```

Dove `tmp_generator.asp` è un file contenente i blocchi con il relativo ID, nella forma `block(ID).`

3.2 Script

Per eseguire il programma `generator.asp` variando le costanti in gioco, è stato creato un semplice script bash che permette di avviare ciclicamente il programma, definire i risultati e salvarli in un file apposito, che potrà essere successivamente utilizzato per il benchmark.

```
#!/bin/bash

# Impostazioni di base
num_files=100
max_width_range=(8 8 8 8 8 8 8 8 8 8)
max_height_range=(8 8 8 8 8 8 8 8 8 8)
max_dim_range=(1 2 2 3 3 4 4 4 5 5)
num_blocks_range=(1 2 3 2 3 2 3 4 2 3)

# Genera i file di benchmark
for file_num in $(seq 1 $num_files); do
    # Calcola l'indice corretto per gli array
    index=$((($file_num-1)/10))

    # Seleziona valori dagli array usando l'indice corretto
    max_width=${max_width_range[$index]}
    max_height=${max_height_range[$index]}
    max_dim=${max_dim_range[$index]}
    num_blocks=${num_blocks_range[$index]}

    # Crea il file temporaneo dei blocchi
    echo "" > tmp_generator.asp
    for ((block=1; block<=num_blocks; block++)); do
        echo "block(b$block)." >> tmp_generator.asp
    done

    # Esegui clingo e salva l'output
    clingo tmp_generator.asp generator.asp \
        --rand-freq=1 \
        --seed=$RANDOM \
        --const max_width=$max_width \
        --const max_height=$max_height \
        --const max_dim=$max_dim \
        | grep -o 'init_block([^)]*)' \
        | sed 's/$/./' > "benchmark$file_num.asp"

    # Aggiungi le costanti al file di output
    echo "#const max_width=$max_width." >> "benchmark$file_num.asp"
    echo "#const max_height=$max_height." >>
    ↪  "benchmark$file_num.asp"
done

rm tmp_generator.asp
```

Pertanto, il programma avrà principalmente tre compiti:

- Generare in un file temporaneo `tmp_generator.asp` i fatti `block(ID)` in base al numero di blocchi definito per il benchmark.
- Eseguire il programma ciclicamente, catturando una configurazione per ogni esecuzione.
- Creare un file di benchmark per ciascuna configurazione generata.

3.3 Risultati benchmark

I benchmark, eseguiti tramite l'apposito script `launcher.sh` per ciascuna configurazione, consentono di analizzare le proprietà emergenti del sistema e il suo comportamento nelle diverse configurazioni.

Dall'analisi dei tempi di esecuzione sono emerse diverse osservazioni:

- All'aumentare del **numero di blocchi**, cresce la complessità del problema e si espande lo spazio di ricerca delle mosse necessarie per raggiungere la soluzione. Tuttavia, questo aspetto è influenzato anche dalla dimensione dei blocchi: blocchi più grandi riducono il numero di mosse disponibili, incidendo sulla complessità della ricerca.
- La **distanza dei blocchi dalla posizione obiettivo** incide significativamente sui tempi di esecuzione. Infatti, una maggiore distanza dal goal implica un numero superiore di mosse necessarie, aumentando il tempo richiesto per la ricerca della soluzione.

In Figura 7 sono riportati due esempi di configurazioni che evidenziano i fenomeni precedentemente descritti. La configurazione mostrata in Figura 6 risulta meno onerosa dal punto di vista computazionale, poiché il numero di mosse necessarie e disponibili è ridotto grazie alla vicinanza al goal e alle dimensioni dei blocchi, consentendo di ottenere una soluzione in circa 10 secondi. Al contrario, in Figura 5, la maggiore distanza dal goal e la diversa dimensione dei blocchi determinano un aumento sia del numero di mosse richieste che del tempo di calcolo, portando a una soluzione in circa 350 secondi.

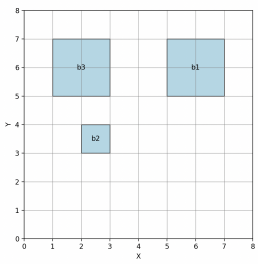

Figure 5: Configurazione difficile

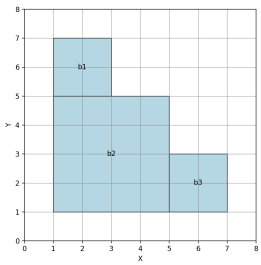

Figure 6: Configurazione comoda

Figure 7: Esempi di configurazioni con livelli di difficoltà differenti

3.4 Configurazioni utilizzate

Per via della complessità del problema e data la limitata potenza di calcolo della macchina sulla quale è stata eseguita, sono state utilizzate le seguenti configurazioni:

- `-parallel-mode=8`: Questa configurazione sfrutta il calcolo parallelo del processore aumentando notevolmente le performance, senza sorprese. In particolare, vengono utilizzati 8 core in modalità `compete`, che permettono di eseguire per ogni thread una configurazione diversa. La configurazione che termina prima ferma la ricerca di soluzioni.
- `-parallel-mode=8,split`: Questa configurazione sfrutta anch'essa il calcolo parallelo del processore ma suddivide il carico di lavoro tra i thread. Aumenta i tempi di risoluzione rispetto alla modalità `compete`, pertanto non si presta bene per questo problema.
- `frumpy`: Configurazione bilanciata, senza ottimizzazioni particolari. Non porta miglioramenti alle performance.
- `handy`: Usa restart più frequenti e decisioni più guidate per problemi difficili. Non porta alcun miglioramento.
- `trendy`: Favorisce l'utilizzo di euristiche. Aumenta i tempi di risoluzione, quindi non è una configurazione utile.
- `crafty`: Strategia avanzata con propagatori specializzati, utile per problemi combinatori complessi. Non porta alcun miglioramento alle perfomance, aumentando i tempi di risoluzione.

3.5 Visualizzazione

Per visualizzare il problema e le mosse che trasformano la configurazione iniziale in quella di goal, è stato utilizzato un generatore di GIF. Questo ha permesso di rappresentare graficamente le mosse eseguite e di individuare eventuali vincoli non rispettati, facilitando così la verifica della correttezza delle soluzioni.

4 CONCLUSIONI E SVILUPPI FUTURI

Dai test effettuati risultano inefficaci le diverse configurazioni offerte dal solver `Clingo` in termini di miglioramento delle performance. Tuttavia, i tempi di risoluzione si mantengono ragionevoli per problemi di media entità e con alcune idee si pensa in futuro di poter aumentare il livello di difficoltà delle configurazioni affrontabili.

La prima idea, potenzialmente quella più efficace, consiste nel fornire a `main.asp` tutte le configurazioni ottime trovate in modo da evitare che esso, durante la ricerca, scarti delle combinazioni di mosse in realtà valide. Col comportamento attuale infatti, vengono scartate anche configurazioni goal poiché diverse dalla singola fornitagli. Questo porterebbe all'aumento delle configurazioni soluzione del problema e quindi di quei casi che porterebbero alla terminazione con soluzione.

Un'altra idea che potrebbe portare un notevole incremento delle prestazioni della ricerca riguarda l'introduzione di un vincolo che si può vedere come evoluzione di quello che evita mosse consecutive di uno stesso blocco opposte. Esso infatti dovrebbe impedire di considerare quelle combinazioni di mosse che portano a muovere uno stesso blocco consecutivamente lungo un percorso che in un qualunque momento risulta tornare in una posizione già visitata (a

meno che nel percorso sia presente una situazione di goal). Durante l'implementazione, è stato testato un vincolo per certi versi simile che forzava ogni mossa a considerare un blocco diverso: questo portò un incremento di performance perché il solver in questo modo non perdeva tempo ad esplorare tanti cammini lunghi di un singolo blocco (tranne nel caso in cui fosse l'utimo a dover raggiungere la posizione goal) ma piuttosto considerava piccoli spostamenti per ciascun blocco. Nella versione finale però questo vincolo è stato rimosso perché rischiava di eliminare sequenze di mosse valide.

Infine, come ulteriore opzione abbiamo l'ampliamento delle configurazioni iniziali riconosciute come impossibili. Infatti, allo stato attuale ci sono alcune configurazioni particolari che non sono immediatamente riconosciute a causa del fatto che vengono considerati non più spostabili blocchi che hanno almeno due lati non opposti completamente occupati. Questo vincolo non considera la situazione particolare mostrata in Figura 8 dove il blocco `b3`, pur avendo un solo lato completamente occupato, è immobilizzato.

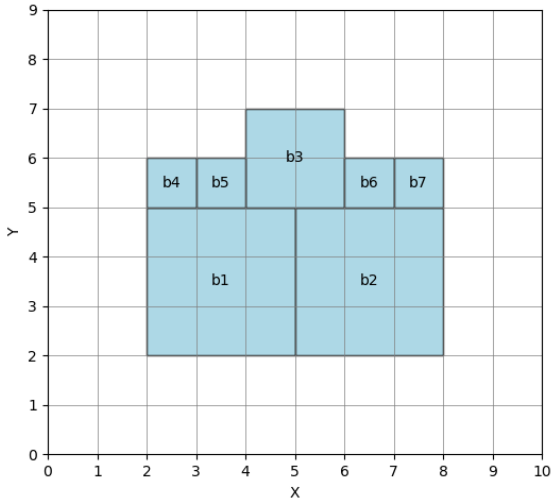

Figure 8: Esempio di configurazione impossibile non considerata attualmente.