Progetto relativo al corso di Automated Reasoning
a.a. 2021-2022

Gianluca Macrì
mat. 142213

Indice

1 Problema assegnato

Il testo originale del problema assegnato è il seguente:

> In una stanza di dimensioni $n \times m$ ci sono degli scatoloni $b_1, \ldots, b_k$. Gli scatoloni sono tutti *quadrati* ma ciascuno ha la propria dimensione del lato (intera) $s_1, \ldots, s_k$. Sono note (dato in input) le posizioni dei vertici più in basso e a sinistra delle scatole.
>
> Un magazziniere ha a disposizione la mossa $\texttt{move}(b_i, d)$ dove d denota la direzione (n,s,w,o). Gli spostamenti sono unitari. Ma è possibile solo *spingere*, non tirare. Dunque ad esempio se un blocco finisce nel bordo destro o sinistro, non potrà più essere spostato orizzontalmente. Una mossa non si può fare se la posizione di arrivo della mossa stessa è già occupata.
>
> L'obbiettivo è quello di ammassare gli scatoloni in basso partendo dall'angolo a sinistra. Se lo spazio nella base non fosse sufficiente si continua ad inserire via via più in alto (stile tetris).
>
> In input viene dato anche un vincolo h che dice che nessun blocco può occupare celle con componente y maggiore di h. Per capirsi con i tipi di blocchi del disegno sotto (immagine 1), se ce ne fossero tanti, sarebbe meglio mettere un blocco piccolo sopra uno piccolo che sopra uno grosso.
>
> Il problema è quello di trovare un piano (una sequenza di mosse) per raggiungere la sistemazione. Possibilmente di lunghezza minima.
>
> Ecco un possibile input, la sua visualizzazione e un possibile output (immagine 1). n=5, m=6, posizioni per le scatole di dimensione 1: $(2,1), (2,4)$, posizione per la scatola di dimensione 2: $(2,3)$.

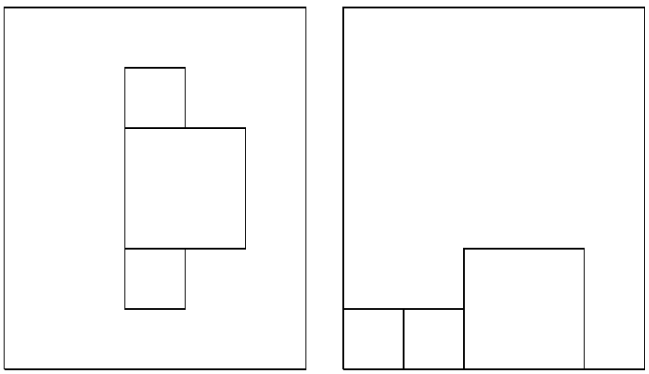

Figura 1: Esempio di un'istanza.

Come si può notare il problema rientra nella categoria dei *planning problem*, nei quali si cerca di trovare un "piano", ossia una sequenza valida di azioni, che permetta di raggiungere uno stato finale a partire dalla configurazione iniziale.

Si riportano quindi una serie di assunzioni che permettono di chiarire ulteriormente la consegna, evitando aspetti potenzialmente ambigui:

1. ciascuna scatola viene identificata dall'indice relativo ad essa (ossia si utilizza k al posto di b_k);
2. invece di utilizzare le direzioni della consegna (n, s, w, o) si impiegano le diciture corrispondenti u (*up*), d (*down*), l (*left*) e r (*right*) per evitare ambiguità con le lettere che denotano la dimensione della stanza;
3. si immagina che il magazziniere abbia bisogno di uno spazio libero di dimensione 1×1 nella parte posteriore della scatola da spingere per ciascuna mossa;
4. si assume che il magazziniere sia in grado di muoversi su tutte le caselle non occupate da scatole, senza essere condizionato da vincoli di raggiungibilità tra esse (magazziniere "volante");
5. si presume che il magazziniere possa spingere solo una scatola alla volta, indipendentemente dalla sua dimensione;
6. la stanza viene assimilata ad una griglia xy $n \times m$ con origine $(1, 1)$ in basso a sinistra e ogni cella viene individuata dalle rispettive coordinate che partono dal valore 1;
7. si formalizza l'ammissibilità di una configurazione finale nel seguente modo: nessuna scatola può occupare una cella con componente y strettamente maggiore di h e, numerando le celle da sinistra a destra, dal basso verso l'alto, detto v_{max} il massimo tra i numeri dei vertici inferiori sinistri tra le diverse scatole, è richiesto che tutte le celle con numerazione minore di v_{max} siano occupate. Si noti come tale formalizzazione, pur rendendo impossibili alcune disposizioni che potrebbero essere desiderabili per certe applicazioni, ad esempio nel caso in cui si disponga solamente di scatole di lato pari da organizzare su più file di una stanza di lato dispari, ha il vantaggio di portare ad una collocazione priva di spazi inutilizzati. Qualora si volessero includere anche tali configurazioni sarebbe necessario utilizzare una proprietà più sofisticata, definendo a priori dei criteri aggiuntivi per specificare come si debbano gestire gli spazi non occupati.

2 Modelli proposti

Per risolvere il problema si sono quindi utilizzati due diversi paradigmi di programmazione: il primo basato sulla *Constraint Programming* ed in particolare utilizzando il tool Minizinc (v2.5.5), ed il secondo basato sull'*Answer Set Programming*, impiegando il solver Clingo (v5.5.1). Inoltre si è pensato di applicare anche due diverse tecniche risolutive.

Nel primo caso, il problema è stato modellato in una versione decisionale, richiedendo di fissare a priori il numero di mosse utilizzabili e cercando una soluzione che impiegasse esattamente tale quantità di spinte. Di conseguenza è stato necessario utilizzare un loop esterno per interrogare il solver con un numero linearmente crescente di mosse, fino al raggiungimento della prima soluzione valida. Tale approccio ha il vantaggio di condurre sempre alla soluzione ottimale, quando questa esiste, ma non permette di rilevare le istanze inammissibili del problema, a meno di non disporre di ulteriori informazioni come ad esempio un limite superiore al numero di mosse. Inoltre, si è notato un notevole rallentamento al crescere del valore ottimale.

Si è quindi deciso di utilizzare anche una modellazione differente che, modificando leggermente il modello decisionale, lo trasformasse in un problema di ottimizzazione. In questo caso è stato necessario definire un limite superiore al numero di mosse richieste, lasciando al solver il compito di cercare delle soluzioni ammissibili via via migliori, fino eventualmente all'ottimo. Tale approccio ha i vantaggi sia di permettere di verificare l'inammissibilità di un'istanza, sia, trovata un prima soluzione conforme ai vincoli, di scegliere quanto tempo aggiuntivo impiegare per cercare di migliorarla, ma non risulta applicabile qualora non fosse possibile limitare il numero di mosse.

Si passa ora alla descrizione dei modelli realizzati per ciascun paradigma, concentrandosi principalmente su quello relativo all'approccio decisionale e accennando brevemente alle modifiche necessarie per la sua trasformazione nella versione di ottimizzazione. Per ciascun paradigma si fornirà inoltre una breve descrizione del formato di input utilizzato.

2.1 Minizinc - Constraint Programming

2.1.1 Formato dell'input

L'input necessario ad entrambi i modelli realizzati è costituito dai seguenti parametri:

- `n` e `m` naturali positivi che rappresentano le dimensioni xy della stanza;
- `h` naturale positivo, limite superiore alle coordinate y occupate nella posizione finale;
- `boxNumber` numero di scatole presenti;
- `boxSideLen`, `boxStartX` e `boxStartY` vettori unidimensionali di `boxNumber` elementi, contenti rispettivamente la dimensione del lato e le coordinate xy iniziali di ciascuna scatola;
- `maxT` intero contenente il numero di mosse a disposizione, o il suo limite superiore nel caso di ottimizzazione.

2.1.2 Modello decisionale

Il modello proposto utilizza diversi array di variabili per mantenere le informazioni di interesse, in particolare `boxMoved` e `dirMoved` contengono rispettivamente le informazioni sulla scatola mossa e sulla sua direzione di movimento per ciascun istante temporale. Inoltre si mantengono due array per ricordare la posizione di ogni scatola nel tempo, descritta come nella configurazione iniziale attraverso le coordinate xy della cella inferiore sinistra occupata da ciascuna (`boxesX` e `boxesY`).

In questo caso si è scelto di modellare anche la figura del magazziniere (o *sokoban*) e il concetto di spazio libero nella destinazione di arrivo di ciascuna scatola spostata, come delle "scatole speciali", che nel seguito verranno indicate mediante i corrispettivi indici utilizzati nel codice: `sokoban` e `spaceIndex`. Di conseguenza, la loro posizione è stata mantenuta negli array precedentemente citati, che presentano dei domini "estesi" per includerli (`EXTBOXNUM`). Tale scelta ha permesso di semplificare la scrittura di alcuni vincoli, migliorando talvolta le prestazioni osservate.

Per quanto riguarda le dimensioni di questi due elementi, `sokoban` occupa un'area pari a 1×1, mentre quella di `spaceIndex` dipende dalle mosse effettuate, assumendo i valori $l \times 1$ o $1 \times l$, a seconda che lo spostamento sia verticale o orizzontale, con l dimensione della scatola spinta in quel momento. Per tale ragione si è deciso di utilizzare due array aggiuntivi (`xExtBoxSideLen` e `yExtBoxSideLen`) che mantengono le dimensioni dei lati delle diverse scatole, includendo anche `sokoban` e `spaceIndex`, al variare del tempo.

Per quanto riguarda i domini temporali si sono utilizzati due insiemi diversi: `MAXTIMEINT` che varia da 0 a `maxT` ed include quindi tutti gli istanti temporali, e `MOVETIMEINT` relativo ai soli istanti in cui si effettua un mossa, ossia coincidente con l'intervallo `0..maxT-1`.

```
array [MOVETIMEINT] of var BOXNUMS: boxMoved;
array [MOVETIMEINT] of var DIRS: dirMoved;

array [MAXTIMEINT,EXTBOXNUMS] of var XS: boxesX;
array [MAXTIMEINT,EXTBOXNUMS] of var YS: boxesY;

array [MAXTIMEINT,EXTBOXNUMS] of var SIDELENGTHS: xExtBoxSideLen;
array [MAXTIMEINT,EXTBOXNUMS] of var SIDELENGTHS: yExtBoxSideLen;

% "copy" values in extended arrays
constraint forall(t in MAXTIMEINT)(
  forall(b in BOXNUMS) (xExtBoxSideLen[t,b] = boxSideLen[b] /\
                        yExtBoxSideLen[t,b] = boxSideLen[b])
  /\ yExtBoxSideLen[t,sokoban] = 1 /\ xExtBoxSideLen[t,sokoban] = 1);

constraint forall(b in BOXNUMS)(boxesX[0,b] = boxStartX[b] /\
                                boxesY[0,b] = boxStartY[b]);
```

Si espongono quindi i diversi vincoli utilizzati, spiegandone brevemente lo scopo. La scelta di mantenerne separanti alcuni, che si saprebbero potuti unire visto che condividono uno stesso `let` o un ciclo `forall` comune, è motivata sia dal voler garantire una migliore comprensibilità del codice, sia dai miglioramenti osservati nelle prestazioni.

Per ogni mossa, a seconda della direzione e della scatola coinvolta, si assegnano le coordinate adeguate di `spaceIndex` e le dimensioni che tale spazio deve avere. Ad esempio, volendo muovere una scatola verso l'alto sarà necessario avere uno spazio di dimensioni $l \times 1$, con l dimensione della scatola da muovere, posto esattamente sopra di essa. Il fatto che tali posizioni non vadano a sforare le dimensioni della stanza viene garantito mediante i vincoli di dominio.

```
constraint forall(t in MOVETIMEINT)
  (let {var DIRS: dir=dirMoved[t], var BOXNUMS: boxMov=boxMoved[t],
        var SIDELENGTHS: movedLen=boxSideLen[boxMov],
        var XS: movedX=boxesX[t,boxMov], var YS: movedY=boxesY[t,boxMov]}
   in  if     dir = U then boxesX[t,spaceIndex] = movedX /\
                           boxesY[t,spaceIndex] = movedY+movedLen /\
                           xExtBoxSideLen[t,spaceIndex] = movedLen /\
                           yExtBoxSideLen[t,spaceIndex] = 1

       elseif dir = D then boxesX[t,spaceIndex] = movedX /\
                           boxesY[t,spaceIndex] = movedY-1 /\
                           xExtBoxSideLen[t,spaceIndex] = movedLen /\
                           yExtBoxSideLen[t,spaceIndex] = 1

       elseif dir = R then boxesX[t,spaceIndex] = movedX+movedLen /\
                           boxesY[t,spaceIndex] = movedY /\
                           xExtBoxSideLen[t,spaceIndex] = 1 /\
                           yExtBoxSideLen[t,spaceIndex] = movedLen

       else                boxesX[t,spaceIndex] = movedX-1 /\
                           boxesY[t,spaceIndex] = movedY /\
                           xExtBoxSideLen[t,spaceIndex] = 1 /\
                           yExtBoxSideLen[t,spaceIndex] = movedLen
       endif);
```

Per ogni mossa è necessario verificare che il `sokoban` sia in una posizione tale per cui è in grado di spingere la scatola interessata. Tale vincolo verrà successivamente utilizzato per garantire che una scatola venuta a contatto con uno dei bordi dell stanza non possa più essere allontanata da esso.

```
constraint forall(t in MOVETIMEINT)(
  (let {var DIRS: dir=dirMoved[t], var BOXNUMS: boxMov=boxMoved[t],
        var SIDELENGTHS: movedLen = boxSideLen[boxMov],
        var XS: movedX=boxesX[t,boxMov], var YS: movedY=boxesY[t,boxMov]}
   in
   if     dir = U
   then boxesX[t,sokoban] >= movedX /\ boxesX[t,sokoban]<movedX+movedLen
        /\ boxesY[t,sokoban]= movedY - 1
   elseif dir = D
   then boxesX[t,sokoban] >= movedX /\ boxesX[t,sokoban]<movedX+movedLen
        /\ boxesY[t,sokoban] = movedY + movedLen
   elseif dir = R
   then boxesX[t,sokoban] = movedX-1 /\ boxesY[t,sokoban] >= movedY
        /\ boxesY[t,sokoban] < movedY + movedLen
   else boxesX[t,sokoban] = movedX+movedLen /\ boxesY[t,sokoban]>=movedY
        /\ boxesY[t,sokoban] < movedY + movedLen
   endif);
```

Per ogni istante temporale si vuole evitare che le scatole si sovrappongano, inclusi `sokoban` e `spaceIndex`, garantendo la correttezza delle varie mosse. Pertanto si è utilizzato il vincolo globale `diffn` che, prese in input le coordinate xy dei vertici inferiori sinistri di elementi rettangolari e i vettori con le lunghezze dei loro lati per entrambe le direzioni, assicura che non ci siano sovrapposizioni. Tale vincolo è utilizzato anche per verificare la coerenza della configurazione finale, senza considerare `sokoban` e `spaceIndex`, in quanto non sono previste mosse successive.

```
constraint forall(t in MOVETIMEINT)(
  (let {var DIRS: dir=dirMoved[t], var BOXNUMS: boxMov=boxMoved[t],
        var SIDELENGTHS: movedLen = boxSideLen[boxMov],
        var XS: movedX=boxesX[t,boxMov], var YS: movedY=boxesY[t,boxMov]}
   in if     dir = U then diffn(boxesX[t,EXTBOXNUMS],
     boxesY[t,EXTBOXNUMS], xExtBoxSideLen[t,EXTBOXNUMS],
     yExtBoxSideLen[t,EXTBOXNUMS])
       elseif dir = D then diffn(boxesX[t,EXTBOXNUMS],
         boxesY[t,EXTBOXNUMS], xExtBoxSideLen[t,EXTBOXNUMS],
         yExtBoxSideLen[t,EXTBOXNUMS])
       elseif dir = R then diffn(boxesX[t,EXTBOXNUMS],
         boxesY[t,EXTBOXNUMS], xExtBoxSideLen[t,EXTBOXNUMS],
         yExtBoxSideLen[t,EXTBOXNUMS])
       else                diffn(boxesX[t,EXTBOXNUMS],
         boxesY[t,EXTBOXNUMS], xExtBoxSideLen[t,EXTBOXNUMS],
         yExtBoxSideLen[t,EXTBOXNUMS])  % dir = L
       endif);

constraint diffn(boxesX[maxT,BOXNUMS], boxesY[maxT,BOXNUMS], boxSideLen,
  boxSideLen);
```

Per vincolare la posizione delle varie scatole nel tempo è necessario modellare l'effetto degli spostamenti e l'inerzia di quelle che non vengono mosse. Al fine di semplificare la scrittura del codice si sono sfruttare delle funzioni ausiliarie per calcolare i valori delle nuove posizioni di una scatola in base alla direzione dello spostamento.

```
constraint forall(t in MOVETIMEINT, b in BOXNUMS)(
  if boxMoved[t] = b
```

```
  then % move effect
    boxesX[t+1,b] = newBoxX(boxesX[t,b],dirMoved[t]) /\
    boxesY[t+1,b] = newBoxY(boxesY[t,b],dirMoved[t])
  else % inertia
    boxesX[t+1,b] = boxesX[t,b] /\
    boxesY[t+1,b] = boxesY[t,b]
  endif);

function var XS: newBoxX(var XS:oldPos, var DIRS:dir) =
  if     dir = R then oldPos+1
  elseif dir = L then oldPos-1
  else               oldPos    % dir = U \/ dir = D
  endif;

function var YS: newBoxY(var YS:oldPos, var DIRS:dir) =
  if     dir = U then oldPos+1
  elseif dir = D then oldPos-1
  else               oldPos    % dir = R \/ dir = L
  endif;
```

Nonostante i vincoli precedenti siano sufficienti a garantire la correttezza delle mosse, per ragioni di efficienza si è preferito mantenere il seguente vincolo ridondante, che costituisce un ulteriore controllo alla disponibilità di spazio per poter spostare le varie scatole.

```
constraint forall(t in MOVETIMEINT)(
  (let {var DIRS: dir=dirMoved[t], var BOXNUMS: boxMov=boxMoved[t],
        var SIDELENGTHS: movedLen = boxSideLen[boxMov],
        var XS: movedX=boxesX[t,boxMov], var YS: movedY=boxesY[t,boxMov]}
   in if     dir = U \/ dir = D then movedY + movedLen <= m /\ movedY > 1
      else                          movedX + movedLen <= n /\ movedX >
       1  % dir = L \/ dir = R
      endif);
```

Infine, si sono utilizzati i due seguenti constraint per garantire i requisiti di posizionamento delle varie scatole, richiesti per lo stato finale.

```
constraint forall(b in BOXNUMS)( boxesY[maxT,b]+boxSideLen[b]-1 <= h);

constraint
  let {var 1..m*n: maxPosVetex = max([
   boxesX[maxT,b]+n*(boxesY[maxT,b]-1) | b in BOXNUMS])} in
    forall(x in XS, y in YS where x+n*(y-1)<=maxPosVetex)
      (exists(b in BOXNUMS)
        (x >= boxesX[maxT,b] /\ x < boxesX[maxT,b]+boxSideLen[b] /\
         y >= boxesY[maxT,b] /\ y < boxesY[maxT,b]+boxSideLen[b]) );
```

Dopo aver sperimentato diverse configurazioni risolutive, utilizzando l'annotazione `int_search` e diversi backend, si è scelto di mantenere la strategia di default `solve satisfy`, apparentemente migliore, e di utilizzare Chuffed come solver.

2.1.3 Modello per l'ottimizzazione

Volendo convertire il precedente modello in un problema di ottimizzazione si può definire un'ulteriore variabile, `finalT`, che rappresenterà l'istante in cui viene raggiunta la configurazione finale e che si cercherà di minimizzare. Di conseguenza `maxT` verrà considerato come un limite superiore al numero totale di mosse e l'intervallo temporale `0..maxT` verrà partizionato in tempo effettivamente utilizzato e tempo in eccesso per ciascuna soluzione.

```
var MAXTIMEINT: finalT;
var set of int: USEDTIME = 0..finalT;
var set of int: MOVETIMEINT = 0..finalT-1;
var set of int: UNUSEDTIME = finalT+1..maxT;
```

Non potendo utilizzare domini contenenti variabili come dimensioni degli array, questi rimarranno definiti come descritto per il caso decisionale e si utilizzeranno dei vincoli aggiuntivi per imporre in maniera arbitraria i valori degli istanti temporali appartenenti a `UNUSEDTIME`, prevenendo così eventuali simmetrie.

```
constraint forall(t in UNUSEDTIME)(
  forall(b in EXTBOXNUMS)(boxesX[t,b]=1 /\ boxesY[t,b]=1));

constraint forall(t in finalT..maxT)
  (xExtBoxSideLen[t,spaceIndex] = 1 /\ yExtBoxSideLen[t,spaceIndex] = 1);

constraint forall(t in UNUSEDTIME where t < maxT)(
  boxMoved[t] = 1 /\ dirMoved[t] = U);
```

I restanti constraint rimangono definiti come in precedenza, sostituendo `finalT` a `maxT` per la verifica dei requisiti della configurazione finale.

Riguardo alla strategia risolutiva, dopo alcuni tentativi, si è scelto di applicare l'annotazione `int_search` sulle variabili che definiscono l'istante finale e le mosse da eseguire, come di seguito riportato, e si è utilizzato il solver Chuffed in modalità free search, consentendogli di ignorare le annotazioni qualora non fossero efficaci per una certa istanza.

```
solve :: int_search([finalT]++boxMoved++dirMoved,
                    input_order, indomain_min, complete)
      minimize finalT;
```

2.2 Clingo - Answer Set Programming

2.2.1 Formato dell'input

Anche in questo caso, come per Minizinc, il formato dell'input coincide per i due modelli ed è costituito dai seguenti parametri:

- `n` e `m`, dimensioni xy della stanza;
- `h` imite superiore alle coordinate y occupate nella posizione finale;
- `t` numero di mosse a disposizione, o il suo limite superiore nel caso di ottimizzazione;
- una serie di predicati `boxInit`, uno per scatola, con 4 argomenti ciascuno: l'indice identificativo della scatola, la lunghezza del lato e le coordinate x e y iniziali.

Tali parametri possono essere inseriti in un file a sé, anteponendo la dichiarazione `#const` alle costanti.

2.2.2 Modello decisionale

Sono stati definiti vari predicati di dominio per limitare lo spazio di ricerca del solver e rendere più efficiente la fase di grounding del modello.

```
xNum(1..n). yNum(1..m). boardVal(1..n*m).
time(0..t).
direction(u). direction(d). direction(l). direction(r).

board(X,Y) :- xNum(X), yNum(Y).
```

```
% coordinates' offsets given a sidelength, useful for occupation
boxLenOffset(L,0..L-1) :- box(_,L).

box(B,L) :- boxInit(B,L,_,_).
```

Per tenere traccia della posizione delle scatole nel tempo si utilizza il predicato `boxPos`, univocamente determinato dalla coppia indice di una scatola e istante temporale. Inoltre ci si preoccupa di garantire la correttezza della configurazione iniziale, evitando che ci siano scatole posizionate oltre i limiti della stanza. Tale proprietà sarà in seguito mantenuta dall'ammissibilità delle mosse effettuate.

```
boxPos(B,L,X,Y,0) :- boxInit(B,L,X,Y), board(X,Y).

1{boxPos(B,L,X,Y,T):board(X,Y)}1 :- box(B,L), time(T).

:- not board(Xb+L-1,Yb+L-1), boxPos(B,L,Xb,Yb,0).
:- not board(Xb,Yb), boxPos(B,L,Xb,Yb,0), box(B,L).
```

Si definisce poi `occupied`, derivato da `boxPos`, per tener traccia nel tempo delle posizioni occupate dalle varie scatole nella stanza. Tale predicato permette di definire un vincolo che eviti le sovrapposizioni di più scatole nella medesima cella.

```
occupied(X,Y,B,T) :- X = Xb + Ox, Y = Yb + Oy,
                     boxLenOffset(L,Ox),boxLenOffset(L,Oy),
                     boxPos(B,L,Xb,Yb,T), board(X,Y), time(T).

:- occupied(X,Y,B1,T), occupied(X,Y,B2,T), B1<B2,
   time(T), box(B1,_), box(B2,_), board(X,Y).
```

Si definiscono quindi i predicati `wallContact` e `freeBack`. Questi, per ogni istante di tempo e scatola, indicano rispettivamente se vi sia un contatto con uno dei bordi delle stanza in una certa direzione e se vi sia almeno una cella libera nella parte posteriore per poter spingere la scatola verso la direzione indicata.

```
wallContact(B,r,T) :- X+L-1 = n, boxPos(B,L,X,Y,T),
                      box(B,L), board(X,Y), time(T).
wallContact(B,l,T) :- X = 1, boxPos(B,L,X,Y,T),
                      box(B,L), board(X,Y), time(T).
wallContact(B,d,T) :- Y = 1, boxPos(B,L,X,Y,T),
                      box(B,L), board(X,Y), time(T).
wallContact(B,u,T) :- Y+L-1 = m, boxPos(B,L,X,Y,T),
                      box(B,L), board(X,Y), time(T).

freeBack(B,u,T) :-  not occupied(Xb,Yb,_,T),
                    Xb = X + Ox, boxLenOffset(L,Ox), Yb = Y-1,
                    board(Xb,Yb), board(X,Y), boxPos(B,L,X,Y,T), time(T).
freeBack(B,d,T) :-  not occupied(Xb,Yb,_,T),
                    Xb = X + Ox, boxLenOffset(L,Ox), Yb = Y+L,
                    board(Xb,Yb), board(X,Y), boxPos(B,L,X,Y,T), time(T).
freeBack(B,l,T) :-  not occupied(Xb,Yb,_,T),
                    Xb = X+L, Yb = Y + Oy, boxLenOffset(L,Oy),
                    board(Xb,Yb), board(X,Y), boxPos(B,L,X,Y,T), time(T).
freeBack(B,r,T) :-  not occupied(Xb,Yb,_,T),
                    Xb = X-1, Yb = Y + Oy, boxLenOffset(L,Oy),
                    board(Xb,Yb), board(X,Y), boxPos(B,L,X,Y,T), time(T).
```

Si può quindi definire un predicato `canBeMoved` che indica quali scatole possano essere spinte nelle varie direzioni, senza superare i confini della stanza. Per ciascun istante temporale, ad eccezione dell'ultimo, si impone di effettuare esattamente una mossa. La correttezza di queste è determinata da `canBeMoved` e dai vincoli riguardanti `occupied` che evitano di avere sovrapposizioni di più scatole a seguito dei movimenti.

```
canBeMoved(B,D,T) :- not wallContact(B,D,T), freeBack(B,D,T),
                     box(B,_), direction(D), time(T).

1{move(B,D,T):box(B,_),direction(D)}1 :- time(T), T<t.

:- move(B,D,T), not canBeMoved(B,D,T), box(B,_), direction(D), time(T).
```

É quindi necessario definire l'effetto di ciascuna mossa sulle posizioni delle scatole spostate e modellare l'inerzia per quelle rimanenti.

```
% move effect
boxPos(B,L,X,Y1,T1):- boxPos(B,L,X,Y,T), move(B,u,T), T1=T+1, Y1=Y+1,
                      box(B,_),time(T),time(T1),board(X,Y1),board(X,Y).
boxPos(B,L,X,Y1,T1):- boxPos(B,L,X,Y,T), move(B,d,T), T1=T+1, Y1=Y-1,
                      box(B,_),time(T),time(T1),board(X,Y1),board(X,Y).
boxPos(B,L,X1,Y,T1):- boxPos(B,L,X,Y,T), move(B,l,T), T1=T+1, X1=X-1,
                      box(B,_),time(T),time(T1),board(X1,Y),board(X,Y).
boxPos(B,L,X1,Y,T1):- boxPos(B,L,X,Y,T), move(B,r,T), T1=T+1, X1=X+1,
                      box(B,_),time(T),time(T1),board(X1,Y),board(X,Y).

% inertia
boxPos(B,L,X,Y,T+1) :- not isMoved(B,T), boxPos(B,L,X,Y,T),
                       time(T), board(X,Y), T<t.

isMoved(B,T) :-  move(B,_,T), box(B,_), time(T).
```

Infine vengono imposti dei vincoli che assicurino la correttezza della configurazione finale, così come definita nella sezione 1.

```
:- occupied(_,Y,_,t), Y>h.

finalPos(B,L,X,Y) :- boxPos(B,L,X,Y,t).

finalOccupiedVal(V) :- V=(Y-1)*n+X, occupied(X,Y,_,t).

maxFinalVertexVal(Vmax) :- Vmax = #max{(Y-1)*n+X: finalPos(_,_,X,Y)}.

:- maxFinalVertexVal(Vmax), boardVal(V), V<Vmax, not finalOccupiedVal(V).
```

2.2.3 Modello per l'ottimizzazione

Per convertire il precedente modello in un problema di ottimizzazione possiamo introdurre un nuovo predicato `maxT(T)`. Questo sarà vero per un solo istante temporale (`1{maxT(T): time(T)}1.`) in cui viene raggiunta la configurazione finale.

Dopo aver eseguito il codice su alcune istanze preliminari, si è scelto di limitare solamente il numero di mosse all'intervallo $0..T_{max}$ con T_{max} valore per cui `maxT` risulta vero. Quindi, per imporre la minimizzazione, si è aggiunto il costrutto appropriato.

I controlli che riguardano la correttezza della configurazione finale, invece, possono coincidere con quelli del modello precedente, considerato che una volta raggiunto T_{max} le scatole rimarranno nella stessa posizione a causa dell'inerzia.

```
1{move(B,D,T):box(B,_),direction(D)}1 :- time(T), T<Tmax, maxT(Tmax).

#minimize{T:maxT(T)}.
```

3 Generazione delle istanze e risultati

Per valutare i modelli è stata preparata una batteria di istanze sulla base delle prestazioni osservate nei test preliminari. Nello specifico si è utilizzato uno script che, dati i parametri desiderati, generasse le configurazioni a partire da una disposizione finale accettabile, effettuando le mosse a ritroso in maniera pseudocasuale. Così facendo è stato possibile definire un limite superiore al numero di spinte necessario per ciascun problema, consentendo l'utilizzo dei modelli ad ottimizzazione.

I parametri controllati, riportati anche all'interno del nome di ciascun file, sono i seguenti: `n` e `m` dimensioni xy della stanza; `b` il numero di scatole; `bxl` e `maxL` che denotano rispettivamente il numero di scatole per ciascuna dimensione maggiore di 1 (quelle di dimensione 1×1 possono comparire un maggior numero di volte) e la lunghezza massima dei lati delle scatole (assegnate in maniera linearmente decrescente fino all'esaurimento di esse); `t` il numero di mosse pseudocasuali da effettuare a ritroso dalla configurazione finale. In particolare, per realizzare le istanze di test si sono considerate due dimensioni per le stanze, 6×8 e 8×10, ciascuna contenete 4, 6 o 8 scatole, di cui una di dimensione 2×2 e le rimanenti di lato 1. Per ciascuna combinazione si sono utilizzati 3 limiti al numero di mosse (15, 30 e 45) e si sono generate due istanze per tipologia, arrivando a 36 casi totali. Come si può notare si sono utilizzate esclusivamente delle configurazioni risolvibili, visto che i modelli decisionali realizzati non sono generalmente in grado di riconoscere quelle che non lo sono.

Si è quindi utilizzato ciascuno modello per provare a risolvere le diverse istanze entro un timeout di 10 minuti ciascuna, ottenendo i risultati riportati nella tabella 1. In particolare per ogni caso vengono indicati: il numero di passi minimo trovato dai modelli, che risulta sicuramente ottimo quando determinato dai modelli decisionali o dai modelli ad ottimizzazione qualora terminino entro il tempo limite (in questo caso i risultati sono evidenziati in grassetto); i tempi di esecuzione dei modelli decisionali; il miglior valore trovato ed il tempo impiegato, per ciascuno dei modelli ad ottimizzazione.

Come si può notare dai risultati, per quanto riguarda i modelli decisionali la versione che sfrutta Minizinc è stata in grado di risolvere un numero maggiore di istanze entro il timeout, impiegando dei tempi generalmente inferiori. Le uniche eccezioni sono costituite dalle istanze più piccole, con valore ottimale inferiore a 10, nelle quali Clingo ottiene delle prestazioni migliori. In generale, entrambi i modelli sono stati in grado di gestire entro il timeout solamente le istanze con un numero di mosse piuttosto ridotto, minore di 16. Ciò è dovuto alla crescente difficoltà incontrata all'aumentare del numero di mosse, imputabile alla natura stessa del problema.

Riguardo ai modelli relativi all'ottimizzazione, si può notare come siano stati in grado di risolvere la quasi totalità delle istanze. In particolare i solver sono riusciti a trovare il valore ottimo delle soluzioni negli stessi casi, ed in tempi similari ai modelli precedenti ma, a differenza di questi, hanno saputo fornire una soluzione anche per istanze più complesse. Inoltre, seppur anche in questo caso Clingo abbia impiegato dei tempi leggermente maggiori della variante realizzata mediante la constraint programming, ha permesso di gestire due istanze in più di quest'ultima.

In generale, si può quindi constatare come, nel caso delle istanze testate, l'aggiunta di un valore limite al numero di mosse consentite abbia permesso ai solver di risolvere anche dei problemi più complessi.

Tabella 1: Risultati delle esecuzioni dei modelli. Per ciascuna istanza si riportano il miglior valore trovato dai modelli e i tempi di esecuzione richiesti, entro un timeout di $600s$. Si evidenziano in grassetto i valori di cui i solver hanno accertato l'ottimalità.

istanza	**miglior valore**	**CP**	**ASP**	**CP ottim.**		**ASP ottim.**	
				soluzione	**tempo**	**soluzione**	**tempo**
n6_m8_b4_bxl1_maxL2_t15_0	**9**	5 s	1 s	9	$<$ 1 s	9	1 s
n6_m8_b4_bxl1_maxL2_t15_1	**9**	5 s	1 s	9	1 s	9	2 s
n6_m8_b4_bxl1_maxL2_t30_0	**10**	5 s	1 s	10	1 s	10	6 s
n6_m8_b4_bxl1_maxL2_t30_1	**14**	16 s	89 s	14	17 s	14	159 s
n6_m8_b4_bxl1_maxL2_t45_0	**14**	11 s	237 s	14	7 s	14	212 s
n6_m8_b4_bxl1_maxL2_t45_1	**15**	63 s	timeout	15	36 s	15	timeout
n6_m8_b6_bxl1_maxL2_t15_0	**11**	58 s	46 s	11	18 s	11	20 s
n6_m8_b6_bxl1_maxL2_t15_1	**11**	13 s	23 s	11	19 s	11	17 s
n6_m8_b6_bxl1_maxL2_t30_0	**12**	43 s	78 s	12	100 s	12	87 s
n6_m8_b6_bxl1_maxL2_t30_1	16	timeout	timeout	16	timeout	16	timeout
n6_m8_b6_bxl1_maxL2_t45_0	**15**	27 s	436 s	15	38 s	15	275 s
n6_m8_b6_bxl1_maxL2_t45_1	21	timeout	timeout	21	timeout	21	timeout
n6_m8_b8_bxl1_maxL2_t15_0	**9**	8 s	1 s	9	3 s	9	4 s
n6_m8_b8_bxl1_maxL2_t15_1	**12**	22 s	28 s	12	13 s	12	25 s
n6_m8_b8_bxl1_maxL2_t30_0	18	timeout	timeout	24	timeout	18	timeout
n6_m8_b8_bxl1_maxL2_t30_1	16	timeout	timeout	16	timeout	16	timeout
n6_m8_b8_bxl1_maxL2_t45_0	18	timeout	timeout	41	timeout	18	timeout
n6_m8_b8_bxl1_maxL2_t45_1	17	timeout	timeout	17	timeout	17	timeout
n8_m10_b4_bxl1_maxL2_t15_0	**9**	6 s	5 s	9	2 s	9	3 s
n8_m10_b4_bxl1_maxL2_t15_1	**10**	8 s	9 s	10	4 s	10	6 s
n8_m10_b4_bxl1_maxL2_t30_0	**13**	45 s	242 s	13	53 s	13	232 s
n8_m10_b4_bxl1_maxL2_t30_1	16	timeout	timeout	16	timeout	16	timeout
n8_m10_b4_bxl1_maxL2_t45_0	**15**	20 s	303 s	15	15 s	15	584 s
n8_m10_b4_bxl1_maxL2_t45_1	17	timeout	timeout	17	timeout	17	timeout
n8_m10_b6_bxl1_maxL2_t15_0	**13**	12 s	212 s	13	28 s	13	163 s
n8_m10_b6_bxl1_maxL2_t15_1	**9**	7 s	11 s	9	2 s	9	10 s
n8_m10_b6_bxl1_maxL2_t30_0	20	timeout	timeout	20	timeout	20	timeout
n8_m10_b6_bxl1_maxL2_t30_1	**14**	141 s	timeout	14	280 s	14	timeout
n8_m10_b6_bxl1_maxL2_t45_0	**14**	53 s	timeout	14	102 s	14	timeout
n8_m10_b6_bxl1_maxL2_t45_1	23	timeout	timeout	27	timeout	23	timeout
n8_m10_b8_bxl1_maxL2_t15_0	**12**	131 s	timeout	12	142 s	12	timeout
n8_m10_b8_bxl1_maxL2_t15_1	**9**	15 s	9 s	9	10 s	9	8 s
n8_m10_b8_bxl1_maxL2_t30_0	20	timeout	timeout	20	timeout	20	timeout
n8_m10_b8_bxl1_maxL2_t30_1	21	timeout	timeout	NA	timeout	21	timeout
n8_m10_b8_bxl1_maxL2_t45_0	19	timeout	timeout	23	timeout	19	timeout
n8_m10_b8_bxl1_maxL2_t45_1	39	timeout	timeout	NA	timeout	39	timeout