Progetto di Automated Reasoning
2021-2022

Christian Londero

15 Febbraio 2022

1 Definizione del problema

Si consideri una scacchiera n x n (n è dato in input). Si hanno a disposizione l pezzi a forma di L, s pezzi a forma di quadrato e r pezzi a forma di rettangolo (si veda la figura, la dimensione del rettangolo è 3 x 1, il quadrato è 2 x 2 e il lato lungo della L è 2). l, s, r sono dati in input. L'obbiettivo è di riempire la scacchiera con i pezzi a disposizione in maniera tale da minimizzare le celle vuote/free. Il requisito aggiuntivo è che f celle (date in input) siano già occupate (quindi vietate).
Si veda l'esempio (le celle grigie sono già occupate/vietate).

1.1 Considerazioni

Si presenterà un algoritmo che tenta di minimizzare le celle empty scegliendo da un sottoinsieme dei pezzi. Quindi si può dare in input un numero molto alto di pezzi al fine di "semplificare" la minimizzazione, sebbene si rischia in questo caso di fare utilizzare gli stessi tipi di pezzi quasi ovunque.
Nella sezione dei risultati vedremo infatti che se abbiamo a disposizione un elevato numero di pezzi ($3*l+4*s+3*r \gg n*n$) allora il solving è "facile", rispetto a un numero quasi giusto di pezzi ($3*l+4*s+3*r \simeq n*n$).

2 Soluzione

L'idea utilizzata per la realizzazione dei modelli è la stessa sia per il modello in minizinc che per quello in asp. Parlando a livello non-implementativo, è la seguente: il programma prende in input i seguenti parametri:

- `n` (intero): dimensione della board;
- `l` (intero): numero di L (2x2) disponibili;
- `s` (intero): numero di quadrati (2x2) disponibili;
- `r` (intero): numero di rettangoli (1x3) disponibili;
- `f` (intero): numero di celle vietate/forbidden/già occupate;
- `forbidden` (array): array lungo f di coordinate X, Y (sono dei fatti/facts in asp).

La board è una matrice (array bi-dimensionale) n x n il cui dominio è definito dalla seguente enumerazione: `XXX, EEE, S11, S12, S13, S14, R11, R12, R13, R21, R22, R23, L11, L12, L13, L21, L22, L23, L31, L32, L33, L41, L42, L43`.
Al fine di rendere comprensibile l'enumerazione si osservi la figura qui di seguito.
Quindi ogni cella può assumere uno di quei valori dell'enumerazione, si è quindi proceduto a definire dei vincoli che mantengano le forme corrette.
Ho utilizzato tale "metodo" al fine di evitare sovrapposizioni indesiderate dei pezzi.
E' possibile visualizzare il modello di minizinc a `./main.mzn` e di clingo/ASP a `./main.lp` per ulteriori dettagli implementativi.

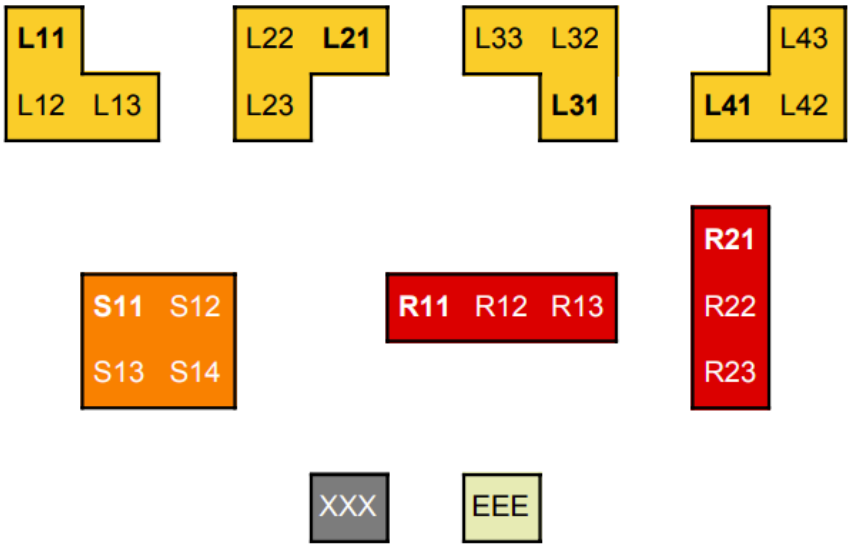

Figure 1: Tutte le possibili shapes con rotazioni

Per minizinc non sono state definite delle annotazioni di ricerca particolari (sebbene siano state tentate ma con migliorie pressochè nulle).

ASP/clingo viene lanciato con i seguenti parametri:

`clingo` *main.lp input* `-t8 --quiet=1,1 --out-hide-aux --outf=2 --time-limit=300 --warn none --configuration=frumpy --opt-strategy=bb,inc`

di cui gli unici di ottimizzazione `-t8 --configuration=frumpy --opt-strategy=bb,inc`; si è infatti evidenziato un notevole miglioramento dei tempi di risoluzione per certe istanze mediante l'ausilio di tali parametri. I restanti parametri sono utili solo a formattare l'output in un modo agevole per essere parsato dal `visualizer` (di cui discuteremo nella sezione 4).
Minizinc viene lanciato con i seguenti parametri:

`minizinc` *main.mzn input* `-O2 --solver  --time-limit 300000 -p12 -f --output-mode json -s --soln-separator ‘‘" --search-complete-msg ‘‘OPTIMUM"`

di cui gli unici di ottimizzazione sono `--O2 -f`. Il primo risulta il miglior compromesso tra tempo di compilazione e guadagno in termini di tempi nella fase di risoluzion; il secondo invece, `-f` (free search), migliora i tempi di molte istanze prese in analisi. I restanti parametri sono utili solo a formattare l'output in modo agevole per il `visualizer`.

2.1 Ulteriori idee

Avevo pensato ad altre due modellazioni che però nell'implementazione poi si sono rivelate errate/inefficienti:

- indicare con 1 le L, con 2 i quadrati e con 3 i rettangoli, con 0 le empty cells e con -1 i forbidden; tuttavia, sebbene il dominio in questo casi sia notevolmente ridotto, con tale metodo non si riescono a gestire efficientemente le sovrapposizioni di forme uguali con diverse rotazioni
- enumerare ogni pezzo mantenendo il tipo salvato in un "array" lungo k con $k = l + s + r$ (per esempio, ignorando le celle empty e forbidden, se abbiamo $k = 2 + 1 + 3$ avremmo che il dominio delle celle della board è $1..k = 1..6$ e manteniamo un ulteriore array lungo k per indicare il tipo: [1,1,2,3,3,3]), in modo tale che il tipo indichi quali constraint il pezzo i-esimo debba avere e non ci sono sicuramente

sovrapposizioni in quanto un pezzo di forma uguale avrebbe un $j! = i$ come "valore" e ogni "valore" può comparire al più 3 o 4 volte nella board a seconda della forma (rispettivamente se L/rettangolo o quadrato); tuttavia questo metodo si è rivelato altamente inefficiente in quanto i domini erano molto più grandi rispetto alla soluzione scelta e portavano quindi a tempi molto più elevati anche con strategie di `int_search` (per minizinc) personalizzate;

3 Utilities

Sono state predisposte alcune utilities per velocizzare la fase di test. In particolare nella folder `./utils/` si trovano:

- `input_gen.py` che prende in input 7 parametri:
  1. q: il numero di istanze (diverse) che vogliamo generare;
  2. n: dimensione della board;
  3. l: numero di L 2x2;
  4. s: numero di quadrati 2x2;
  5. r: numero di rettangoli 1x3;
  6. f: numero di celle proibite (scelte random);
  7. *output_path* (OPZIONALE): path alla cartella dove verranno inseriti gli input generati; qualora non specificato, crea gli input salvandoli nella folder `../inputs` (che viene creata in automatico).

  Genera sia il `dzn` che il `lp`. Un esempio di esecuzione: `python utils/input_gen.py 1 10 10 7 10 10 10`

- `run_tests.py` che prende in input 2 parametri:
  1. *input_path* (OPZIONALE): path alla cartella dove verranno cercati gli input da lanciare; qualora non specificato, cerca gli input nella folder `../inputs`;
  2. *output_path* (OPZIONALE): path alla cartella dove verranno inseriti gli output generati; qualora non specificato, crea gli output salvandoli nella folder `../outputs` (che viene creata in automatico).

  esegue i rispettivi visualizer a seconda del tipo di file in input. Genera anche un file `metadata.csv` all'interno dell'*output_path* che contiene le info circa l'esecuzione di ciascun solver (qualora non ci fossero errori). Un esempio di esecuzione:
  `python utils/run_tests.py`

4 Visualizer delle soluzioni

Per agevolare la visualizzazione dell'output dei solver (sia asp che minizinc) è stato predisposto un `visualizer`, splittato per clingo e minizinc, scritto in python che riceve in input il main, l'input e opzionalmente dove salvare l'output e restituisce un file html che renderizza la griglia trovata dal solver (se non restituisce errori) in formato HTML + CSS.
Più tecnicamente, per stampare l'output dell'input i di clingo/asp salvato nella folder ./inputs/input$i$.lp (o di minizinc salvato nella folder ./inputs/input$i$.dzn), si può lanciare il seguente comando:

```
python visualizer/asp_visualize.py main.lp inputs/inputi.lp outputs/aspi.html
python visualizer/mzn_visualize.py solver main.mzn inputs/inputi.dzn outputs/mzn-solveri.html
```

e aprendo il file `outputs/aspi.html` (o mzn-*solveri*.html) si potrà visualizzare la soluzione visivamente con le relative stats. Per minizinc *solver* può essere *gecode* oppure *coinbc*.
Sia minizinc che ASP all'interno dei rispettivi visualizer vengono eseguiti con dei parametri customizzati

per restituire gli output in json per poi essere utilizzati comodamente dal visualizer. I parametri di ottimizzazione invece vengono descritti nella sezione successiva.
Il terzo parametro è opzionale. Qualora non fosse specificato, l'output viene salvato automaticamente nella folder `outputs`.
I colori dei pezzi sono:

- L: giallo
- quadrato: arancione
- rettangolo: rosso
- empty cell: bianca
- forbidden cell: grigia

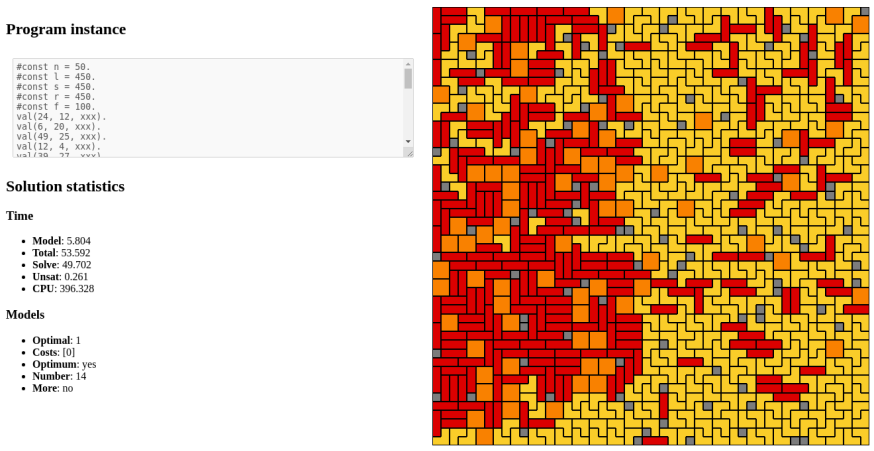

Figure 2: Esempio del visualizer sull'input 12 di ASP/clingo

5 Risultati sperimentali

I test sono stati effettuati su un PC con OS Ubuntu 18.04, con 8GB di RAM e CPU intel core i7 4770HQ@2.20GHz.
Il modello di ASP risulta molto più performante rispetto a quello di minizinc per la maggior parte degli input analizzati. La suddivisione in istanze easy/medium/hard verrà fatta in base al tempo minore di risoluzione tra minizinc e ASP.
Per minizinc sono stati studiati i comportamenti dei solver gecode e coinbc. Il solver coinbc risultava particolarmente più efficace rispetto a gecode per il modello definito.
Si noti che il modello di minizinc restituisce un "out of memory" da $n = 50$ in poi, sebbene comunque risulti "affaticato" già con istanze aventi un n minore.
D'altrocanto il modello ASP fatica a "confermare" l'ottimo trovato con istanze aventi (un numero di) pezzi "quasi giusti" che riescono a ricoprire l'area copribile ($n * n - f$), si veda infatti l'input numero 4. Il problema invece non si pone per istanze aventi un numero di pezzi insufficienti a coprire l'intera area copribile, in quanto è stato definito un lower bound al "minimo" di celle empty (`main.lp:40`).

E' possibile visualizzare ogni input/output dei risultati proposti di seguito (come output dei rispettivi `visualizer`) nella cartella `./examples/outputs`. Per ogni output c'è il prefisso relativo al solver/metodo utilizzato (asp, minizinc-gecode, minizinc-coinbc). Qualora non fosse disponibile un output, il motivo è che il solver è andato in out of memory.
I costi vengono evidenziati in grassetto qualora l'output sia un ottimo.

5.1 Istanze "easy"

Per istanza easy del problema (relativa alla soluzione proposta) si intende o un problema con n relativamente piccolo, oppure un problema con n non molto grande ma avendo a disposizione k pezzi con $area(k) \gg n*n-f$ (quindi l'area di tutti i pezzi disponibili "eccessivamente" maggiore rispetto all'area copribile). Di seguito una tabella che riassume alcuni casi "easy" con f celle scelte a random.

	inputs					minizinc - gecode		minizinc - coinbc		clingo (asp)	
#	n	l	s	r	f	cost	time (s)	cost	time (s)	cost	time (s)
1	6	5	4	5	5	**0**	0.427	**0**	0.884	**0**	0.016
2	7	6	5	6	6	**0**	0.574	**0**	1.136	**0**	0.024
3	8	7	6	7	8	**0**	1.565	**0**	1.446	**0**	0.029
4	10	10	8	10	10	**1**	1.633	**1**	2.975	1	*timeout*
5	10	11	9	11	10	**0**	1.870	**0**	2.578	**0**	0.073
6	11	15	11	15	20	**0**	22.856	**0**	2.768	**0**	0.066
7	12	16	14	16	20	**0**	9.064	**0**	3.596	**0**	0.088
8	20	150	100	120	55	125	*timeout*	**0**	14.325	**0**	0.370
9	25	150	100	120	60	380	*timeout*	**0**	94.784	**0**	1.455
10	30	150	100	120	70	528	*timeout*	0	*timeout*	**0**	4.307
11	40	300	300	300	150	1156	*timeout*	20	*timeout*	**0**	20.006

Figure 3: Easy instances

5.2 Istanze "medium"

Per istanza media del problema (relativa alla soluzione proposta) si intende un problema con n relativamente grande (tra 50 e 60) avendo a disposizione k pezzi con $area(k) \gg n*n-f$ (quindi l'area di tutti i pezzi disponibili "eccessivamente" maggiore rispetto all'area copribile). Di seguito una tabella che riassume alcuni casi "medium" con f celle scelte a random.

	inputs					minizinc - gecode		minizinc - coinbc		clingo (asp)	
#	n	l	s	r	f	cost	time (s)	cost	time (s)	cost	time (s)
12	50	450	450	450	100	-	-	-	-	**0**	53.592
13	51	450	450	450	100	-	-	-	-	**0**	57.570
14	52	450	450	450	100	-	-	-	-	**0**	65.477
15	53	450	450	450	100	-	-	-	-	**0**	73.717
16	54	450	450	450	100	-	-	-	-	**0**	82.942
17	55	450	450	450	100	-	-	-	-	**0**	91.049
18	56	450	450	450	100	-	-	-	-	**0**	99.423
19	57	450	450	450	100	-	-	-	-	**0**	113.958
20	58	450	450	450	100	-	-	-	-	**0**	123.311
21	59	450	450	450	100	-	-	-	-	**0**	140.527
22	60	450	450	450	100	-	-	-	-	**0**	154.571

Figure 4: Medium instances

5.3 Istanze "hard"

Per istanza hard del problema (relativa alla soluzione proposta) si intende o un problema con n grande ($n > 60$) con (un numero di) pezzi "quasi giusti" che riescono a ricoprire l'area copribile ($n * n - f$) oppure un problema con $n > 70$. Di seguito una tabella che riassume alcuni casi "hard" con f celle scelte a random.

	inputs					minizinc - gecode		minizinc - coinbc		clingo (asp)	
#	n	l	s	r	f	cost	time (s)	cost	time (s)	cost	time (s)
23	61	450	400	400	100	-	-	-	-	57	*timeout*
24	62	450	400	400	100	-	-	-	-	62	*timeout*
25	63	450	400	400	100	-	-	-	-	55	*timeout*
26	64	450	400	400	100	-	-	-	-	70	*timeout*
27	65	450	450	450	100	-	-	-	-	77	*timeout*
28	66	450	450	450	100	-	-	-	-	76	*timeout*
29	67	450	400	400	100	-	-	-	-	-	-
30	70	500	500	500	100	-	-	-	-	-	-
31	71	500	500	500	100	-	-	-	-	-	-
32	72	500	500	500	100	-	-	-	-	-	-

Figure 5: Hard instances