Pràctica Scala + SAT: Crowded Chessboard
Programació Declarativa: Aplicacions: 2023

Aniol Molero Grau i Bernat Comas Machuca

18 de gener de 2024

Índex

1 Introducció

Aquesta pràctica consisteix a resoldre el problema del *Crowded Chessboard*, força semblant al conegut problema de les N Reines, on cal col·locar un nombre N de reines en un tauler d'escacs d'NxN, d'una manera que cap reina s'amenaci entre si. En aquest cas, el *Crowded Chessboard* té la dificultat afegida que no només tenim reines, sinó que també tenim torres, alfils, cavalls i reis. Les peces s'han de col·locar de manera que no amenacin cap peça del mateix tipus, sense tenir en compte les peces que hi pugui haver entre elles.

Per resoldre aquest problema hem fet servir Scala, i hem utilitzat la llibreria ScalAT, que se'ns ha proporcionat amb l'enunciat d'aquesta pràctica. Després d'acabar de completar els mètodes no implementats de la llibreria, l'hem feta servir per poder representar el problema com una fórmula CNF (*Conjunctive Normal Form*), generada a partir de les diferents clàusules que restringeixen les posicions on pot anar cada tipus de peça.

Com hem dit, les diferents clàusules de la CNF les hem fet amb la llibreria ScalAT, que ens permet aplicar fàcilment els *cardinality constraints* AMO (*At Most One*), ALO (*At Least One*) i EO (*Exactly One*), i també les seves variacions AMK (*At Most K*), ALK (*At Least K*) i EK (*Exactly K*), ja siguin les versions amb encoding quadràtic com logarítmic.

En aquesta pràctica tenim diferents instàncies, és a dir, diferents taulers i conjunts de peces predefinits, començant amb un tauler de mida 5x5, i acabant amb un tauler de mida 16x16, cada vegada amb un nivell de complexitat més elevat. Cal dir que en aquest problema és molt més complex i costós trobar la insatisfactibilitat d'una instància que trobar-ne una distribució que la satisfaci, ja que, si és satisfactible, només cal trobar una distribució correcta, mentre que si no és satisfactible, cal explorar tot l'espai de cerca per trobar que, efectivament, no hi ha cap distribució vàlida. Això fa que, arribats a cert nivell de complexitat, ens sigui impossible poder arribar a trobar la instasifacibilitat d'una instància en menys d'una hora, que és el *timeout* que se'ns proposa.

2 Descripció dels diferents cardinality constraints implementats

Cadascun dels següents *cardinality constraints* es tracta d'una funció que rep una llista de variables sobre les quals cal aplicar la restricció corresponent, i també un valor K si és necessari.

2.1 At Most One amb encoding logarítmic

Per implementar aquesta part hem fet que, donada una llista de més d'una variable (ja que si només tenim una variable, un *At Most One* no restringeix la variable de cap manera), representem cada variable com un nombre binari de N o menys bits, on N és el logaritme en base 2 de la mida de la llista, arrodonit a l'alça, és a dir, el nombre de bits necessaris per representar la variable d'índex més gran. Cada vegada que convertim una variable la seva codificació en binari, anem afegint les clàusules corresponents al valor de cada bit. Per cada bit, si el bit és 1, afegim la clàusula $(\neg x(i) \vee y(j))$, on x(i) és una de les variables de la llista inicial de variables, i y(j) és un dels bits dels nombres. Pel cas en què el bit és 0 o quan la variable no necessita el bit per ser representada, afegim la clàusula $(\neg x(i) \vee \neg y(j))$.

```
def addAMOLog(x: List[Int]): Unit = {
    if(x.length <= 1)
      return

    val n: Int = Math.ceil(Math.log(x.length)/log2).toInt
    val y: List[Int] = newVarArray(n).toList

    // Iterate over all the variable indexes in x
    for (i <- x.indices) {
      val bin: String = i.toBinaryString
      val offset = n - bin.length

      //Negate variables not present in the binary number
      for (j <- 0 until offset) {
        addClause(-x(i) :: -y(j) :: List())
      }

      //Handle each case (0 or 1 values)
      for (j <- offset  until n) {
        if (bin(j - offset) == '0') {
          addClause(-x(i) :: -y(j) :: List())
        }
        else {
          addClause(-x(i) :: y(j) :: List())
        }
      }
    }
}
```

La primera clàusula és simplement transformació de la implicació lògica $x(i) \rightarrow y(j)$, mentre que la segona és la transformació de $x(i) \rightarrow \neg y(j)$.

2.2 At Most K amb sorting networks

En aquest cas, si el valor K és més gran que la mida de la llista de variables, no cal fer res, ja que, encara que poséssim totes les variables a cert, no ens passaríem d'aquest

límit.

Ara bé, quan no es dona aquesta situació, cal crear una variable auxiliar y_i per cada variable x_i que tinguem. Llavors, passem la llista de variables originals i la llista de variables auxiliars al *sorter* per relacionar-les tenint en compte l'ordre de les variables, començant amb totes les certes i acabant amb totes les falses. Finalment, només cal afegir la clàusula $(\neg y(K))$ per assegurar que, com a molt, hi ha K variables certes. Cal tenir en compte que el primer índex és el 0, de manera que això farà que només de les variables auxiliars y(0) a y(K-1) puguin ser certes, cosa que influirà sobre els valors que poden agafar les diferents variables x.

```
def addAMK(x: List[Int], K: Int): Unit = {
    if (K<0) { // Si la llista s buida est implicit
      // Error
      return
    }
    else if(K >= x.length) {
      // Satisfet
      return
    }

    val y: List[Int] = newVarArray(x.length).toList

    addSorter(x, y)

    addClause(-y(K) :: List())
}
```

2.3 At Least K amb sorting networks

El procés per aquest *cardinality constraint* és pràcticament el mateix que l'anterior. Ignorem els casos en què la llista és buida o la K és 0 o menys, però ara hem de tenir en compte que, si la K és més gran que la mida de la llista de variables, no es podrà complir mai. Per tant, si passa això creem una variable auxiliar aux, i afegim les clàusules (aux) i $(\neg aux)$. Clarament, és impossible que es compleixin les dues clàusules simultàniament, cosa que provocarà que tota la CNF sigui insatisfactible.

A banda d'això, fem el mateix de crear variables auxiliars i relacionar-les amb les variables originals, però ara afegim la clàusula (y(K-1)), per forçar que, tenint les variables ordenades, com a mínim K variables siguin certes (de 0 a K-1).

```
def addALK(x: List[Int], K: Int): Unit = {
    if(x.isEmpty || K <= 0)
      return
    else if(K>x.length) {
      val aux: Int = newVar();
      addClause(aux :: List())
      addClause(-aux :: List())
    }
    val y: List[Int] = newVarArray(x.length).toList

    addSorter(x, y)

    addClause(y(K-1) :: List())
}
```

2.4 Exactly K amb sorting networks

Una vegada hem fet els altres dos casos, aquest és molt senzill d'implementar. Simplement, cal fer tant *At Most K* com *At Least K* sobre la llista de variables, ja que si forcem que volem com a molt K i com a mínim K variables certes, l'única manera que això es compleixi és tenir exactament K variables certes, és a dir, *Exactly K*.

```
def addEK(x: List[Int], K: Int): Unit = {
    addALK(x, K)
    addAMK(x, K)
}
```

3 Explicació del model

3.1 Viewpoint utilitzat

Pel nostre model, hem utilitzat un *viewpoint* on tenim una matriu NxN variables per cada tipus de peça, on N és la mida del tauler. La idea és que la matriu de variables per cada peça estigui a cert si a la posició corresponent del tauler hi ha una peça d'aquell tipus. D'aquesta manera, evitem que hi hagi les simetries que tindríem amb un *viewpoint* que ens digués a quina casella va cada peça, ja que en el nostre cas només indiquem quin tipus de peça hi ha a cada casella. En cas que cap variable de cap matriu sigui certa per una casella, significarà que la casella és buida.

A banda d'això, també tenim dues variables que indiquen si a una diagonal o a una contradiagonal hi ha un alfil. Veient els valors de les diferents instàncies ens vam adonar que, en un tauler de mida N, hi tenim 2*(N-1) alfils, i també tenim 2*(N-1) diagonals i 2*(N-1) contradiagonals. A més, per posar tots els alfils d'una instància del *Crowded Chessboard*, cal que totes les diagonals excepte una i totes les contradiagonals excepte una estiguin ocupades per un alfil.

3.2 Restriccions

3.2.1 Restriccions bàsiques

En primer lloc, hem d'assegurar-nos que a una mateixa casella només hi ha, com a molt, una sola peça. Òbviament, pot no haver-hi res, de manera que utilitzem un *At Most One* per aquest cas:

```
    for (i <- 0 until n)
      for (j <- 0 until n) {
        val l = reines(i)(j) :: torres(i)(j) :: alfils(i)(j) :: cavalls(i
    )(j) :: List()
        if (configuracioLog) e.addAMOLog(l)
        else e.addAMOQuad(l)
      }
```

Seguidament, per a què una solució sigui vàlida, de cada tipus de peça se n'han de col·locar exactament el nombre indicat per la instància, ni més, ni menys. Per tant, farem servir un *Exactly K* on, per la restricció de cada peça, la K serà el nombre de peces d'aquell tipus que hi ha a instància que volem resoldre:

```
    e.addEK(reines.flatten.toList, nReines)
    e.addEK(torres.flatten.toList, nTorres)
    e.addEK(alfils.flatten.toList, nAlfils)
    e.addEK(cavalls.flatten.toList, nCavalls)
```

Tot això assegura que al tauler hi tenim totes les peces necessàries per trobar una solució vàlida, i que mai tindrem més d'una peça a una mateixa casella. És clar, però, encara falta fer que cap peça amenaci a cap altra peça del mateix tipus.

Com que les reines es poden moure en totes les direccions, i sempre que tenim un tauler de mida NxN tenim N reines, és fàcil veure que a cada fila i a cada columna hi ha d'haver exactament una reina, perquè, si no, o bé no les estaríem posant totes, o bé s'estarien amenaçant entre elles. A més a més, tant a les diagonals com a les contradiagonals hi pot haver com a molt una sola reina, encara que no és necessari que n'hi hagi cap:

```
    //A cada fila hi ha una reina
    for (i <- reines) {
      if (nReines >= n) {
        if (configuracioLog) e.addEOLog(i.toList)
        else e.addEOQuad(i.toList)
      }
      else {
        if (configuracioLog) e.addAMOLog(i.toList)
        else e.addAMOQuad(i.toList)
      }
    }
```

```
    //A cada fila hi ha una reina
    for (i <- reines) {
      if (nReines >= n) {
        if (configuracioLog) e.addEOLog(i.toList)
        else e.addEOQuad(i.toList)
      }
      else {
        if (configuracioLog) e.addAMOLog(i.toList)
        else e.addAMOQuad(i.toList)
      }
    }
```

```
    //A cada contradiagonal hi ha com a molt una reina
    for (v <- 0 to 2 * n - 2) {
      val l = (for (i <- 0 until n; j <- 0 until n; if i + j == v) yield
    reines(i)(j)).toList
      if (configuracioLog) e.addAMOLog(l)
      else e.addAMOQuad(l)
    }
```

```
    //A cada diagonal hi ha com a molt una reina
    for (v <- -n + 1 until n) {
      val l = (for (i <- 0 until n; j <- 0 until n; if i - j == v) yield
    reines(i)(j)).toList
      if (configuracioLog) e.addAMOLog(l)
      else e.addAMOQuad(l)
    }
```

Després, com que les torres només es poden moure en vertical i horitzontal, i els alfils només es poden moure per les diagonals i contradiagonals, les restriccions que cal afegir per cada tipus de peça tenen la mateixa estructura que la de les reines.

Per les torres tenim el següent:

```
    //A cada fila hi ha una torre
    for (i <- torres) {
      if (nTorres >= n) {
        if (configuracioLog) e.addEOLog(i.toList)
        else e.addEOQuad(i.toList)
      }
      else {
        if (configuracioLog) e.addAMOLog(i.toList)
        else e.addAMOQuad(i.toList)
      }
    }
```

```
    //A cada columna hi ha una torre
    for (i <- torres.transpose) {
      if (nTorres >= n) {
        if (configuracioLog) e.addEOLog(i.toList)
        else e.addEOQuad(i.toList)
      }
      else {
        if (configuracioLog) e.addAMOLog(i.toList)
        else e.addAMOQuad(i.toList)
      }
    }
```

I pels alfils:

```
    //A cada contradiagonal hi ha com a molt un alfil
    for (v <- 0 to 2 * n - 2) {
      val l = (for (i <- 0 until n; j <- 0 until n; if i + j == v) yield
    alfils(i)(j)).toList
      if (configuracioLog) e.addAMOLog(l)
      else e.addAMOQuad(l)
    }
```

```
    //A cada diagonal hi ha com a molt un alfil
    for (v <- -n + 1 until n) {
      val l = (for (i <- 0 until n; j <- 0 until n; if i - j == v) yield
    alfils(i)(j)).toList
      if (configuracioLog) e.addAMOLog(l)
      else e.addAMOQuad(l)
    }
```

Finalment, pels cavalls simplement hem creat una llista dels moviments possibles que poden fer, i hem forçat que a cada parella de caselles on dos cavalls es podrien amenaçar, hi hagi com a molt un cavall. Hem fet una petita optimització en què, en comptes de tenir tots els moviments possibles, només en tenim una meitat, ja que les restriccions faltants ja es posaran per les caselles anteriors. Si hem afegit la restricció que entre les caselles A8 i B6 només hi pot haver com a molt un cavall, no cal afegir la mateixa restricció a la inversa, és a dir, entre B6 i A8:

```
    val possibleMoves = List((1, -2), (-1, -2), (2, -1), (-2, -1))
    for (x <- 0 until n; y <- 0 until n)
      for (i <- possibleMoves; if i._1 + x >= 0 && i._2 + y >= 0 && i._1
    + x < n && i._2 + y < n)
        if (configuracioLog) e.addAMOLog(List(cavalls(x)(y), cavalls(i._1
     + x)(i._2 + y)))
        else e.addAMOQuad(List(cavalls(x)(y), cavalls(i._1 + x)(i._2 + y)
    ))
    }
```

3.2.2 Restriccions implicades

Vam veure que un tauler de NxN té 2*(N-1) diagonals i 2*(N-1) contradiagonals, i que a totes les instàncies del *Crowded Chessboard* hi ha un alfil a cada diagonal i contradiagonal. Per tant, vam pensar que potser afegir la restricció que totes les diagonals i contradiagonals han de tenir un alfil podria ajudar a accelerar el procés del solucionador. Vam decidir fer dues reificacions, una entre els alfils que hi haguessin a cada casella i una *array* de variables que indica si la diagonal x està ocupada o no, i una altra reificació igual però per les contradiagonals.

La implicació bidireccional que hem fet ha sigut $Diagonal_i \leftrightarrow Casella1_i \vee Casella2_i \vee ... \vee CasellaN_i$, és a dir, que si la $Diagonal_i$ és certa, llavors alguna de les caselles que formen la diagonal també és certa, i viceversa.

Si apliquem les transformacions corresponents, ens queda la implicació $Diagonal_i \rightarrow Casella1_i \vee Casella2_i \vee ... \vee CasellaN_i$ i la implicació $Casella1_i \vee Casella2_i \vee ... \vee CasellaN_i \rightarrow Diagonal_i$.

La primera es pot traduir directament per la clàusula $\neg Diagonal_i \vee Casella1_i \vee Casella2_i \vee ... \vee CasellaN_i$.

Per la segona, primer obtenim la clàusula $\neg(Casella1_i \vee Casella2_i \vee ... \vee CasellaN_i) \vee Diagonal_i$, que passa a ser $(\neg Casella1_i \wedge \neg Casella2_i \wedge ... \wedge \neg CasellaN_i) \vee Diagonal_i$, que, després d'aplicar factor comú, es converteix en $(\neg Casella1_i \vee Diagonal_i) \wedge (\neg Casella2_i \vee Diagonal_i) \wedge ... \wedge (\neg CasellaN_i \vee Diagonal_i)$, és a dir, tot de clàusules separades pels operadors $\wedge$, que haurem d'afegir al problema.

A més a més, també especificarem la restricció que hem dit al principi, que totes les diagonals i contradiagonals, és a dir, exactament 2*(N-1) per cada tipus, han d'estar ocupades:

```
1    if (nAlfils >= 2 * (n - 1)) {
2      //Reifiquem caselles amb alfil i diagonals
3      for (v <- -n + 1 until n) {
4        var casellesDiag = (for (i <- 0 until n; j <- 0 until n; if i - j
     == v) yield alfils(i)(j)).toList;
5        for (i <- casellesDiag.indices) e.addClause(-casellesDiag(i) ::
    diagonals(v + n - 1) :: List())
6        e.addALO(List(-diagonals(v + n - 1)).concat(casellesDiag))
7      }
8
9      //Hi ha d'haver exactament 2*(n-1) diagonals amb un alfil
10     e.addEK(diagonals.toList, 2 * (n - 1))
11
12     //Reifiquem caselles amb alfil i contradiagonals
13     for (v <- 0 to 2 * n - 2) {
14       var casellesContradiag = (for (i <- 0 until n; j <- 0 until n; if
      i + j == v) yield alfils(i)(j)).toList;
15       for (i <- casellesContradiag.indices) e.addClause(-
    casellesContradiag(i) :: contradiagonals(v) :: List())
16       e.addALO(List(-contradiagonals(v)).concat(casellesContradiag))
17     }
18
19     //Hi ha d'haver exactament 2*(n-1) diagonals amb un alfil
20     e.addEK(contradiagonals.toList, 2 * (n - 1))
21   }
```

Amb aquesta millora es redueix molt el temps que es tarda tant a trobar una solució com el que es tarda a determinar que una instància és insatisfactible.

Un altre fet interessant del qual ens vam adonar després d'anar fent proves és que, degut al nombre d'alfils que tenim a cada instància, totes les solucions tenen tots els alfils als extrems del tauler. Per tant, si forcem que cap alfil pugui estar a cap casella ïnterna", es redueix molt l'espai de cerca, disminuint el temps que es tarda a trobar una solució, encara que el temps necessari per determinar que una instància no és satisfactible no varia massa:

```
1     // Nom s hi ha alfils als extrems del tauler
2     if(nAlfils == 2*(n-1)) {
3       for (i <- 1 until n - 1) {
4         for (j <- 1 until n - 1) {
5           e.addClause(-alfils(i)(j) :: List())
6         }
7       }
8     }
```

3.3 Trencament de simetries

Per trencar simetries i ajudar a reduir l'espai de cerca, la primera manera que vam pensar va ser, sabent que a cada fila hi ha una reina, forçar que la reina de la primera fila ha d'estar en una fila menor que la reina de l'última fila. Ho hauríem pogut fer al revés, o fer-ho per columnes, però deixant de banda la naturalesa aleatòria en què s'explora l'espai de cerca, els resultats serien més o menys els mateixos. Amb aquesta restricció tan senzilla ja trenquem les simetries verticals i horitzontals per separat, encara que la simetria vertical i horitzontal de manera combinada es manté, o el que és el mateix, una rotació de 180^{o}. A més a més, també tenim les simetries en fer rotacions de 90^{o} i -90^{o}.

Tot i ser una millora important, vam voler trencar encara més simetries, i ens vam adonar que, el fet que els alfils han d'estar sempre als extrems del tauler, i que totes les diagonals excepte una i totes les contradiagonals excepte una han d'estar ocupades, implica que forçosament dues de les quatre cantonades del tauler estan sempre ocupades. Si a això li sumem que els alfils no poden estar a la mateixa diagonal o contradiagonal, podem veure que, clarament, els alfils de les cantonades han d'estar a la mateixa fila o a la mateixa columna per no amenaçar-se. Per tant, una altra restricció que vam afegir va ser que a la cantonada de dalt a l'esquerra i la de baix a l'esquerra (coordenades 0,0 i n-1,0) hi ha d'haver un alfil, mentre que a les altres dues no. Altra vegada, hi hauria diferents combinacions a escollir, però quant a la reducció de l'espai de cerca seria força semblant escollir que els alfils es trobin a unes cantonades o unes altres.

Seguidament, mostrem una imatge de com podem veure gràficament aquestes simetries, amb l'exemple del tauler de 5x5:

Figura 1: Simetries del problema. En verd l'única opció permesa.

Com podem veure, aplicant combinacions de simetries verticals i horitzontals, i rotacions podem obtenir solucions simètriques que, tot i estrictament ser distribucions diferents, essencialment són la mateixa solució. Ja només per la restricció dels alfils que, com hem dit, imposa que hi hagi un alfil a la coordenada 0,0, i un altre alfil a la coordenada n-1,0 veiem que només tenim dues solucions possibles, les de més a l'esquerra de la imatge. Llavors, gràcies a la restricció que obliga que la reina de la primera fila estigui més a l'esquerra que la reina de l'última fila, només ens queda una opció disponible, la que està enquadrada en verd.

Cal dir que en alguns casos, el fet de reduir l'espai de cerca però augmentar la complexitat de les restriccions, pot provocar un augment del temps per trobar una solució, és a dir, demostrar satisfactibilitat, però vam considerar que era un canvi que valia la pena.

4 Experiments

A continuació mostrarem els resultats empírics que hem obtingut de la nostra codificació. En primer lloc compararem els encodings quadràtic i logarítmic per demostrar satisfactibilitat, i a continuació compararem els resultats de les millores que hem fet respecte a la versió bàsica que haviem desenvolupat inicialment, que no inclou restriccions implicades. Per veure els resultats empírics de les diferents execucions vegeu la secció 6.

4.1 Comparació entre encodings per trobar SAT

L'encoding que s'utilitza per codificar els AMOs té un impacte molt significatiu en el temps d'execució de la pràctica. En la següent taula es pot veure el temps d'execució per cada instància de tauler a demostrar satisfactibilitat a la versió base (sense part extra) però amb la codificació millorada.

Chessboard Size	Quad	Log
5	0,03328	0,022
6	0,06137	0,04077
7	0,28268	0,16046
8	0,27448	1,0906
9	0,46842	8,62232
10	4,76313	6,51261
11	24,35495	14,47162
12	6,029761	23,08616
13	509,2307	374,73980
14	357,6869	93,72995
15	-	1.025,31640
16	-	-

Figura 2: Temps d'execució per les codificacions quadràtiques i logarítmiques

Figura 3: Gràfic de comparació de temps d'execució entre l'encoding quadràtic d'A-MOs i l'encoding logarítmic.

Tal i com podem veure la part logarítmica és millor que la quadràtica en instàncies complicades. Això era el que esperavem, ja que el nombre de clàusules generades passa de ser quadràtic a ser logarítmic, i per tant és normal que el SAT solver tardi menys. El mateix passarà si intentem demostrar insatisfactibilitat.

4.2 Comparació entre la versió bàsica i la millorada

En aquesta part, entenem com a versió bàsica la que només té en compte les restriccions que hem especificat a l'apartat de restriccions, al subapartat de restriccions bàsiques, que vénen a ser els *constraints* que simplement forcen que es col·loquin totes les peces,

i que eviten que aquestes s'amenacin entre sí. La versió millorada seria la que té totes les restriccions que hem mencionat, és a dir, restriccions bàsiques, restriccions implicades, i restriccions per trencar simetries.

Chessboard Size	Bàsic (SAT)	Bàsic (UNSAT)	Millorat(SAT)	Millorat(UNSAT)
5	0,306253	0,182717	0,022	0,02
6	0,12488	1,278182	0,04077	0,15109
7	0,8605	16,50792	0,16046	1,24297
8	0,342561	102,814883	1,0906	3,1115
9	238,09835	604,44969	8,62232	15,85384
10	289,98664		6,51261	72,290889
11			14,47162	151,19073
12			23,08616	1633,6078
13			374,73980	
14			93,72995	
15			1.025,31640	

Figura 4: Taula de comparació de temps d'execució entre l'encoding bàsic i el millorat.

Tal i com podem veure a la figura 4 l'encoding que hem creat millorat redueix significativament els temps, tant de la part satisfactible com insatisfactible. A la figura 5 podem veure gràficament la comparació dels temps d'execució segons si utilitzem un encoding bàsic o millorat i segons la satisfactibilitat.

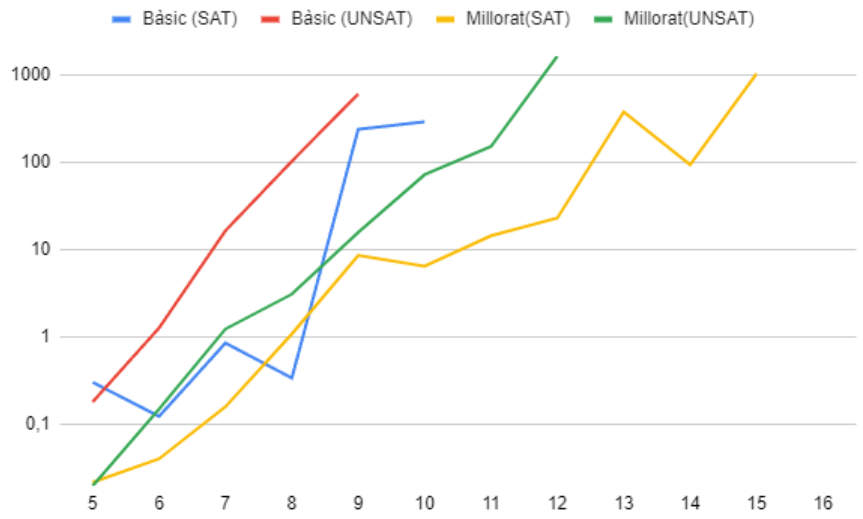

Figura 5: Gràfic de comparació de temps d'execució entre l'encoding bàsic i el millorat i satisfactibilitat i insatisfactibilitat.

Tal i com podem veure el que ens dona uns temps pitjors és l'insatisfactible bàsic, i seguidament demostrar satisfactibilitat amb els constraints bàsics i insatisfactibilitat en els millorats, que segueixen una progressió molt semblant. Cal deixar clar que en el gràfic l'eix vertical és logarítmic perquè ens interessava més poder comparar individualment cada instància amb les diferents execucions que les instàncies entre elles, i altrament no hagués estat possible comparar a la vegada les instàncies petites i les grans degut a que es mouen en diferents ordres de magnitud.

5 Part extra

La part extra d'aquesta pràctica consisteix en, donada una instància, determinar el nombre màxim de reis que permeti que aquesta continuï essent satisfactible.

La primera idea va ser mirar de resoldre la instància amb un rei, i després anar provant d'afegir un rei més cada vegada fins que el problema sigui insatisfactible. Si trobem que no hi ha solució amb N reis, vol dir que el nombre màxim de reis que podem posar és N-1. D'aquesta manera es pot trobar fàcilment el nombre màxim de reis, però s'ha de tenir en compte que, si per exemple a una instància s'hi poden posar 20 reis, vol dir que haurem de resoldre el problema 20 vegades, més una última vegada per trobar insatisfactibilitat amb 21 reis.

Llavors, una idea que vam creure que seria millor va ser primer trobar una solució sense reis, i veure quants reis es poden posar a la distribució obtinguda, sense moure cap peça. Si bé generalment aquesta no serà la distribució òptima que ens permetrà posar el màxim nombre de reis, la podrem utilitzar per aconseguir molt ràpidament una cota inferior del nombre de reis que podem afegir a la instància. Si agafem l'exemple anterior d'una instància on s'hi poden posar 20 reis, si a la solució inicial ja n'hi podem posar 10, vol dir que podem començar fent proves amb 11 reis, cosa que estalvia moltes execucions.

També cal afegir que, evidentment, la cota superior del nombre de reis que podem afegir és la quantitat de caselles buides al tauler, és a dir, el nombre de caselles del tauler menys la quantitat de peces que cal posar a la instància (sense tenir en compte els reis, és clar).

Llavors, primer de tot, ens guardem la solució inicial, sense reis, que hem trobat anteriorment. Seguidament, calculem el nombre de caselles buides que hi ha. Després, creem un nou objecte ScalAT que utilitzarem per obtenir la cota mínima, és a dir, el nombre màxim de reis que queben a la solució inicial, i crearem la matriu de variables que indiquen si a una casella hi ha un rei o no. A continuació, obtenim la llista de caselles que estan ocupades per altres peces:

```
1     actualitzarAuxiliars(e, false) //Guardem la soluci  trobada
2     val emptySpaces = n*n - nReines - nAlfils - nTorres - nCavalls //
      Calculem el nombre de caselles buides que hi ha
3     e2 = new ScalAT("CrowdedChessboardKings") //Creem un nou objecte
      ScalAT que utilitzarem per obtenir la cota m nima
4     reis = e2.newVar2DArray(n,n)
5     //Llista de caselles ocupades per altres peces
6     val l = (for(i <- 0 until n; j <- 0 until n; if(
7       e.getValue(reines(i)(j)) |  e.getValue(torres(i)(j)) |  e.getValue(
      alfils(i)(j)) |  e.getValue(cavalls(i)(j))
8     )) yield reis(i)(j)).toList
```

Quan hem fet tot això, ja podem començar a calcular el nombre de reis mínim. Després d'inicialitzar el nombre de reis a provar a 0, només cal fer un bucle on al principi es creï una nova instància de ScalAT, juntament amb la matriu de variables per les posicions on hi ha un rei, s'augmenti el nombre de reis a provar en una unitat, s'afegeixin les restriccions necessàries i s'intenti trobar una solució.

En aquest cas, les restriccions són que no hi hagi cap rei a una casella ocupada per una altra peça, que hi hagi exactament nReis, on nReis és el nombre de reis que provarem de posar a cada volta del bucle, i totes les restriccions necessàries per evitar que els reis s'amenacin, igual que com s'ha fet amb els cavalls, però amb una llista de moviments diferents.

Aquest bucle s'anirà repetint fins que no s'aconsegueixi trobar una solució amb nReis reis, cosa que significarà que com a mínim podem fer servir nReis-1 reis, o fins que es trobi una manera d'emplenar totes les caselles amb reis, de manera que tant el mínim com el màxim nombre de reis seria nReis, i no caldria fer res més.

```
    //Obtenim el nombre de reis m xim en la soluci  inicial,  s  a dir,
     el  m  nim  nombre de reis que podem posar a la inst ncia
    var nReis = 0
    var result2: SolverResult = null
    var kingsTime: Double = 0
    do {
      e2 = new ScalAT("CrowdedChessboardKings")
      nReis += 1
      reis = e2.newVar2DArray(n, n)
      for (i <- 1.indices) e2.addClause(-1(i) :: List()) //No hi pot
    haver cap rei a una casella ocupada per una altra pe a
      e2.addEK(reis.flatten.toList, nReis) //La soluci  ha de contenir
    nReis reis
      for (x <- 0 until n; y <- 0 until n) {
        for (i <- possibleMovesReis; if i._1 + x >= 0 && i._2 + y >= 0 &&
     i._1 + x < n && i._2 + y < n) {
          //Restriccions sobre les caselles que els reis poden amena ar
          if (configuracioLog) e2.addAMOLog(List(reis(x)(y), reis(i._1 +
    x)(i._2 + y)))
          else e2.addAMOQuad(List(reis(x)(y), reis(i._1 + x)(i._2 + y)))
        }
      }

      result2 = e2.solve()
      kingsTime += result2.time
      if (result2.satisfiable) {
        println("* Determinat que com a m nim  s  satisfactible amb " ++
     nReis.toString() ++ " rei(s) *")
        for (i <- reines.indices) {
          for (j <- reines.indices) {
            reisAux(i)(j) = e2.getValue(reis(i)(j)) //Guardem la soluci
     trobada (la part dels reis nom s)
          }
        }
      }
    } while (result2.satisfiable & nReis < emptySpaces)
```

Una vegada tenim el nombre mínim de reis que podem posar a la instància, cal anar provant de solucionar el problema original amb un rei més cada vegada, fins que o bé trobem un nombre de reis que faci que sigui insatisfactible, o bé emplenem totes les caselles buides amb reis.

Finalment, fem un bucle que s'executi mentre no hàgim trobat el màxim nombre de reis, és a dir, mentre no trobem una instància insatisfactible, i mentre no hàgim col·locat tants reis com caselles buides. A cada volta del bucle es crea una nova instància de ScalAT, i se li afegeixen tots els constraints de la part obligatòria i els constraints concrets de la part extra, utilitzant nReis reis. Llavors simplement cal mirar de solucionar la instància i depenent de si és satisfactible o no, tornar-ho a provar amb un rei més, o mostrar l'última solució satisfactible obtinguda:

```
    var maximNombreTrobat = false
    while(nReis < emptySpaces && !maximNombreTrobat) {
      e = new ScalAT("CrowdedChessboard")

      println("Provem-ho amb " ++ nReis.toString ++ " reis:")
      afegirConstraintsReis(e, nReis) //Afegim totes les restriccions de
    la part obligat ria  m  s les restriccions dels reis, per nReis

```

```
      result2 = e.solve()
      kingsTime += result2.time
      println("King's time: " ++ kingsTime.toString)

      if (result2.satisfiable) {
        println("Satisfactible. Provem-ho amb un rei m s.\n")
        actualitzarAuxiliars(e, true) //Actualitzem la soluci  trobada
        nReis += 1
      }
      else {
        maximNombreTrobat = true //Com que amb nReis no ha sigut satisfactible, vol dir que ja hem trobat el nombre m xim (nReis-1)
        println("No  s  satisfactible.\n")
        println("Per tant, el m xim nombre de reis que podem posar s n" ++ (nReis - 1).toString ++ " rei(s):\n")
      }
    }

    getTauler
```

Els temps d'execució del programa són els que es poden veure a la figura **??**

Chessboard Size	Extra
5	0,0211
6	0,18
7	2,531
8	12,4915
9	103,7337
10	410,683
11	1783,7849
12	-
13	-
14	-
15	-
16	-

Figura 6: Taula de temps d'execució del màxim de reis que podem afegir a un tauler.

Com podem veure en aquest cas els temps augmenten molt ràpidament degut a que tenim divergències entre el lower bound calculat omplint el tauler on no hi ha reis i el valor actual. A més per comprovar que és el màxim cal demostrar que un nombre de reis és insatisfactible, el que augmenta enormement el temps.

6 Resultats obtinguts per diferents instàncies

A continuació tenim les solucions que hem obtingut d'algunes instàncies, tant en el cas en què no posem cap rei com en el cas de la part extra:

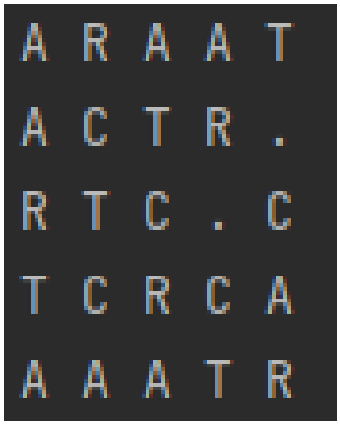

Figura 7: Inst. 0: Distribució sense reis

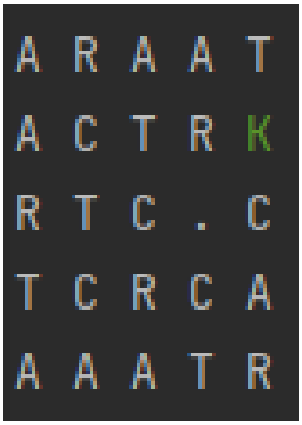

Figura 8: Inst. 0: Distribució amb reis

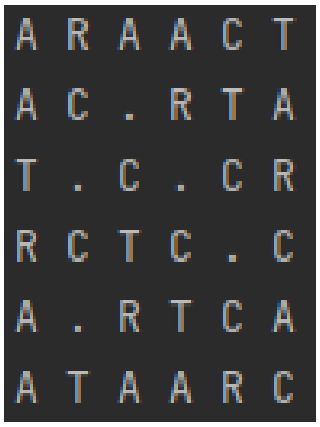

Figura 9: Inst. 1: Distribució sense reis

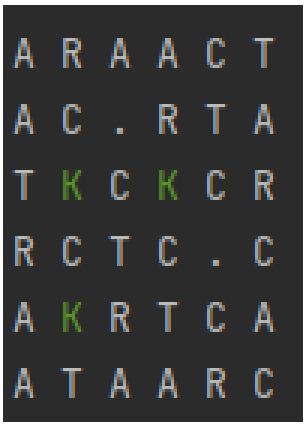

Figura 10: Inst. 1: Distribució amb reis

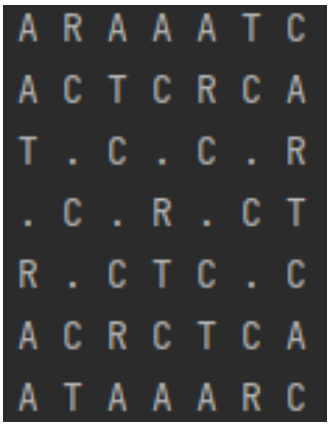

Figura 11: Inst. 2: Distribució sense reis

Figura 12: Inst. 2: Distribució amb reis

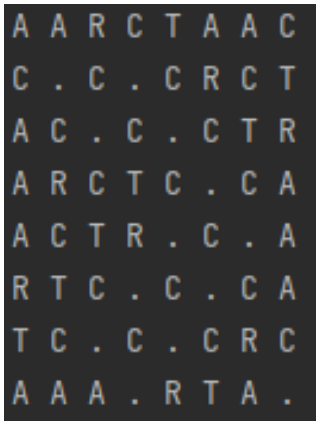

Figura 13: Inst. 3: Distribució sense reis

Figura 14: Inst. 3: Distribució amb reis

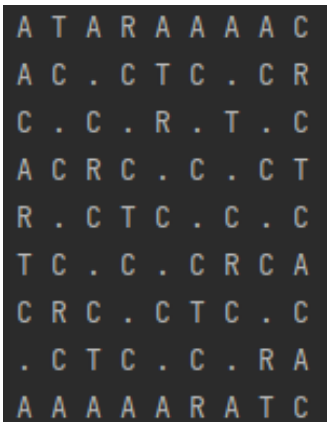

Figura 15: Inst. 4: Distribució sense reis

Figura 16: Inst. 4: Distribució amb reis

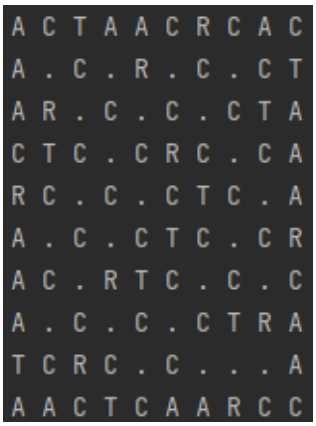

Figura 17: Inst. 5: Distribució sense reis

Figura 18: Inst. 5: Distribució amb reis

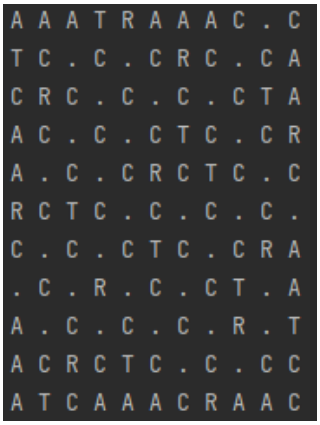

Figura 19: Inst. 6: Distribució sense reis

Figura 20: Inst. 6: Distribució amb reis

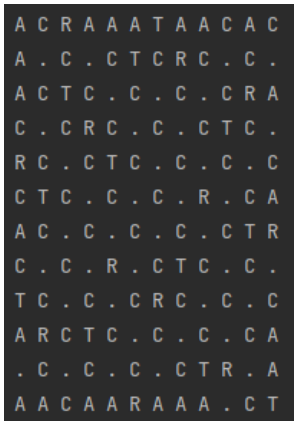

Figura 21: Inst. 7: Distribució sense reis

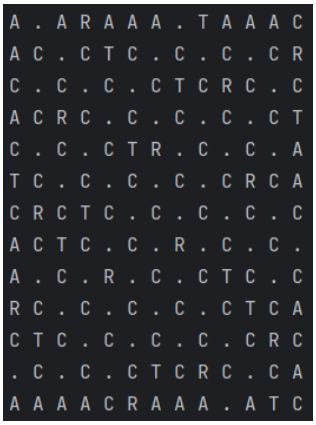

Figura 22: Inst. 8: Distribució sense reis

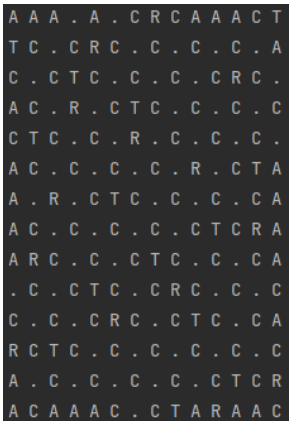

Figura 23: Inst. 9: Distribució sense reis

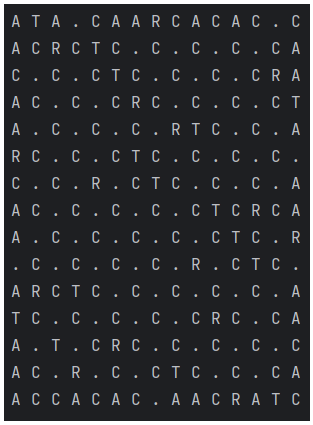

Figura 24: Inst. 10: Distribució sense reis

Responent a una de les preguntes que es fan a la pràctica, en la solució del tauler de

5x5 de l'exemple (que de fet, és la mateixa que generem nosaltres) hi ha dos espais buits i només s'hi pot posar un rei, però no hi ha cap manera de fer que aquests espais quedin més separats per tal de poder-hi posar dos reis.

En les altres instàncies, podem veure que, a vegades, la distribució que obtenim per poder posar el nombre màxim de reis canvia respecta de la solució obtinguda a la part obligatòria (sense tenir en compte els reis és clar), ja que la primera solució trobada no era l'òptima per a maximitzar el nombre de reis. Amb això és fàcil veure que, deixant de banda solucions simètriques o rotades, les instàncies poden tenir més d'una solució vàlida.

7 Conclusions i suggeriments

Primerament, una cosa que ens en vam adonar en començar la pràctica, quan estàvem comprovant que haguéssim fet bé les codificacions dels diferents *cardinality constraints*, i que podem veure en els apartats anteriors, va ser que els *encodings* quadràtics eren una mica més ràpids que els logarítmics per les instàncies més petites, potser perquè, gràcies a la seva simplicitat, eren una mica més fàcils de computar. Ara bé, a mesura que anem augmentant la mida del tauler, cada cop la codificació logarítmica agafa més força davant la quadràtica. Per tant, creiem que la millor opció, en general, és fer servir un *encoding* logarítmic, encara que en alguns casos concrets en què sabem que haurem de resoldre moltes instàncies petites podria valdre la pena utilitzar la codificació quadràtica.

A banda d'això, una altra cosa que ens hem adonat és que normalment reduir l'espai de cerca implica augmentar el temps necessari per trobar una solució, i viceversa. La majoria de les restriccions que hem afegit per trencar simetries han fet que fos més fàcil trobar que una instància no té solució, però també dificulten trobar-ne alguna si no n'hi ha. A més, hi ha el problema afegit que a vegades és complicat de quantificar fins a quin punt un canvi millora o empitjora cada cas, ja que les instàncies són diferents i la forma en què es van trobant les solucions no és una cosa que a priori puguem predir, de manera que per veure si un canvi és bo o no, cal fer proves amb diverses instàncies i després valorar què preferim.

Un altre fenomen molt curiós que hem observat és que algunes instàncies que tenen un costat de tauler parell tarden menys que les que tenen el costat imparell i són més petites. Es pot veure en les execucions entre la instància 13 i 14 i en alguns casos entre la 7 i la 8 o la 9 i la 10. Primer ens havíem plantejat si aquest fet podria tenir alguna relació amb què la mida del tauler sigui parella o no, però finalment atribuïm aquest fenomen a les particularitats del SAT solver i la manera que té d'explorar l'arbre de cerca, ja que en alguns casos aconseguirà arribar a una solució qualsevol més fàcilment. Podem veure que això no passa mai en els casos insatisfactibles, perquè independentment de l'ordre en què recorri l'espai de cerca, s'haurà d'explorar tot.

Quant a suggeriments, per aquesta part de la pràctica no en tenim cap. Ens ha semblat una pràctica correcta, tant de temari com de dificultat i llargada, i ens ha semblat força interessant haver d'anar pensant maneres d'aconseguir restringir cada vegada més el problema. Sobretot ha estat bé la part d'haver de trobar restriccions poc òbvies, com que els alfils han d'estar tots als extrems del tauler, o els dos trencaments de simetries que hem mencionat anteriorment.

8 Bibliografia

Referències

[1] Varis, *Scala Basics* (Enllaç a la pàgina)

[2] Mateu Villaret et al., *Privat. Apunts de classe: Programació Declarativa. Aplicacions.*