Constraints i SAT

Programació Declarativa, Aplicacions
20 de gener del 2025

Wilber Eduardo Bermeo Quito
Dr. Mateu Villaret Auselle
Universitat de Girona

Jordi Badia Auladell
41591544T
jordibadiauladell@gmail.com

Aniol Juanola Vilalta
41559862N
u1978893@campus.udg.edu

Continguts

1. Part Minizinc

El problema s'ha plantejat des de tres *viewpoints* diferents:

1. Tauler indexat per dia i estadi → primera versió que facilita la comprensió del model.
2. Tauler indexat per equip i equip → segon *viewpoint* que busca accedir a la variable més ràpid i controlar les simetries i duplicitats sense redundàncies.
3. Dues matrius indexades per equip i equip → variant del segon *viewpoint* per experimentar si es pot optimitzar l'emmagatzament i accés a dades.

1.1. *Viewpoint* 1

Aquest ha estat el primer *viewpoint* implementat que consisteix en plasmar d'una forma pràcticament literal el que diu l'enunciat. És per això que hi ha moltes restriccions que fan que l'execució sigui molt lenta. Així mateix, facilita la lectura i comprensió del codi i del model.

1.1.1. Variables

- **`tauler[DAY, STADIUM]`**: matriu on cada posició conté un conjunt de TEAM de cardinalitat 0 o 2.
  - 0: no hi ha partit en aquell estadi aquell dia.
  - 2: els dos equips que jugaran en aquell estadi aquell dia.
- **`distanciesEquips[TEAM]`** (auxiliar): variable per representar la distància total recorreguda per cada equip.
- **`seguidorsFora[DAY, STADIUM]`** (auxiliar): nombre de seguidors que no poden accedir a l'estadi per falta de capacitat.

1.1.2. Dominis

- **`tauler[DAY, STADIUM]`**: var set of TEAM amb 0 o 2 elements.
- **`distanciesEquips[TEAM]`** (auxiliar): en base als *lower bounds* i *upper bounds* (Secció 1.1.4.)
- **`seguidorsFora[DAY, STADIUM]`** (auxiliar): en base als *lower bounds* i *upper bounds* (Secció 1.1.4.)

1.1.3. Restriccions

No s'han utilitzat restriccions globals, però s'han utilitzat les següents:

- variant `all_different` amb reïficacions: per assegurar que no es repeteixin partits[1].

```
constraint forall(t1, t2 in TEAM where t1 < t2)(
    sum([t1 in tauler[d, s] /\ t2 in tauler[d, s] | d in DAY, s in STADIUM]) == 1
);
```

- `sum`: per comptar partits assignats per estadi/dia i validar les assignacions.

```
constraint forall(d in DAY, t in TEAM) (
    sum([bool2int(t in tauler[d,s]) | s in STADIUM]) == 1
);
constraint forall(s in STADIUM, t in TEAM) (
    sum([bool2int(t in tauler[d,s]) | d in DAY]) == 1
);
```

- Assignació d'equips fixes preestablerts amb `forall` i verificació de pertinença.

```
constraint forall(d in DAY, s in STADIUM)(
    if fixes[d, s] != {} then
        fixes[d, s] subset tauler[d, s]
```

[1]Hem decidit no utilitzar `all_diferent` i utilitzar **reïficacions** ja que no podem fer un `all_diferent` d'un set.

```
    endif
);
```

- Restricció de cardinalitat de 0 o 2 per a cada cel·la amb la funció `card`.

```
constraint forall(d in DAY, s in STADIUM) (
    card(tauler[d,s]) in {0,2}
);
```

- Garantir un nombre fix de partits per dia i per estadi.

```
constraint forall(d in DAY)(
    sum([card(tauler[d, s]) > 0 | s in STADIUM]) == nmachesperday
);
constraint forall(s in STADIUM)(
    sum([card(tauler[d, s]) > 0 | d in DAY]) == nmatchesperstadium
);
```

1.1.3.1. Restriccions per a funcions objectiu

- Minimitzar km recorreguts per l'equip (mitjançant la variable auxiliar `distanciesEquips`)

```
constraint
    forall(t in TEAM) (
        distanciesEquips[t] = sum([
            distancies[st1, st2] | d in 1..ndays-1, st1 in STADIUM, st2 in STADIUM
            where t in tauler[d, st1] /\ t in tauler[d+1, st2]
        ])
    );

constraint distTotal = sum(distanciesEquips);
```

- Maximitzar entrades de seguidors: (i.e. minimitzar seguidors a fora, amb `seguidorsFora`)

```
constraint
    forall(d in DAY, s in STADIUM)(
        seguidorsFora[d, s] =
            if card(tauler[d, s]) == 0
            then 0
            else max(sum([tifosi[t] | t in tauler[d, s]]) - capacitats[s], 0)
        endif
    );

constraint seguidorsForaTotal = sum(seguidorsFora);
```

1.1.4. *Lower bound* i *Upper bound*

- Minimitzar km recorreguts per l'equip (mitjançant la variable auxiliar `distanciesEquips`)

```
int: lboundDistance = min([distancies[s,s2] | s, s2 in STADIUM where s != s2]) *
(ndays - 1);
int: uboundDistance = max([distancies[s,s2] | s, s2 in STADIUM where s != s2]) *
(ndays - 1);
```

```
array[TEAM] of var lboundDistance..uboundDistance: distanciesEquips;
var lboundDistance*nteams..uboundDistance*nteams: distTotal;
```

- Maximitzar entrades de seguidors: (i.e. minimitzar seguidors a fora, mitjançant la variable auxiliar `seguidorsFora`)

```
int: lboundSeguidors = 0;
int: uboundSeguidors = max(0, max(tifosi)*2 - min(capacitat));

array[DAY, STADIUM] of var lboundSeguidors..uboundSeguidors: seguidorsFora;
var       lboundSeguidors*ndays*nstadiums..uboundSeguidors*ndays*nstadiums:
seguidorsForaTotal;
```

1.1.5. Restriccions implicades i simetries

Com a restricció implicada hem vist que hi ha exactament un partit per equip al dia i a l'estadi.

Pel que fa a les simetries, han quedat trencades usant un set pels equips, de manera que no importa l'ordre i no hi ha duplicats.

1.2. *Viewpoint* 2

En aquest cas, s'ha pres com a variable principal una matriu que indexa per dos equips (que jugaran entre ells). Això donarà lloc a una simetria marcada per la diagonal de la matriu (es veurà més endavant). En cada posició de la matriu hi haurà el dia i l'estadi on es produirà l'encontre.

1.2.1. Variables

- **`tauler[TEAM, TEAM]`**: matriu on cada posició conté una tupla amb el dia i lloc on es jugarà el partit.
- **`team_distance[TEAM]`** (auxiliar): variable per representar la distància total recorreguda per cada equip.
- **`spectators_left_out[TEAM, TEAM]`** (auxiliar): nombre de seguidors que no poden accedir a l'estadi per falta de capacitat.

1.2.2. Dominis

- **`tauler[TEAM, TEAM]`**: var tuple(DAY, STADIUM).
- **`team_distance[TEAM]`** (auxiliar): en base als *lower bounds* i *upper bounds* (Secció 1.1.4.)
- **`spectators_left_out[TEAM, TEAM]`** (auxiliar): en base als *lower bounds* i *upper bounds* (Secció 1.1.4.)

1.2.3. Restriccions

- Trencament de simetries

```
constraint forall(t in TEAM)(
    tauler[t, t] = (1, 1)
);

constraint forall(t, t2 in TEAM where t < t2)(
    tauler[t, t2] = tauler[t2, t]
);
```

- Programació de partits fixats

```
constraint forall(d in DAY, s in STADIUM)(
    if card(fixes[d, s]) = 1 then
        exists(t2 in TEAM where t2 != fixes[d, s][1]) (tauler[fixes[d, s][1], t2] =
(d, s))
    elseif card(fixes[d, s]) = 2 then
        tauler[fixes[d, s][1], fixes[d, s][2]] = (d, s)
    else
        true
    endif
);
```

- Unicitat de dies i estadis per equip

```
constraint forall(t in TEAM)(
    all_different([tauler[t, t2].1 | t2 in TEAM where t2 != t]) /\
    all_different([tauler[t, t2].2 | t2 in TEAM where t2 != t])
);
```

- Codificació per aconseguir valors únics de combinació dia-estadi

```
constraint all_different([
    tauler[t1, t2].1 * 100 + tauler[t1, t2].2
    | t1, t2 in TEAM where t1 < t2
]);
```

1.2.3.1. Restriccions per a funcions objectiu

- Càlcul de distàncies recorregudes

```
constraint forall(t in TEAM) (
    team_distance[t] = sum(d in 1..ndays-1)(
        let {
            var STADIUM: prev_stadium;
            var STADIUM: curr_stadium;
             constraint exists(t2 in TEAM where t != t2)(tauler[t, t2] = (d,
prev_stadium));
            constraint exists(t2 in TEAM where t != t2)(tauler[t, t2] = (d+1,
curr_stadium));
        } in
        distancies[prev_stadium, curr_stadium]
    )
);
```

- Càlcul d'espectadors fora

```
constraint forall(t1, t2 in TEAM where t1 < t2)(
    spectators_left_out[t1, t2] = max(
        tifosi[t1] + tifosi[t2] - capacitats[tauler[t1, t2].2],
        0
    )
);
```

1.2.4. *Lower bound* i *Upper bound*

Els mateixos que en el model anterior (Secció 1.1.4.).

1.2.5. Cerca de solució optimitzada

```
solve :: seq_search([
    int_search([team_distance[t] | t in TEAM], input_order, indomain_min),
   int_search([spectators_left_out[t1, t2] | t1, t2 in TEAM where t1 < t2], input_order,
indomain_min),
    int_search([tauler[t1, t2].1 * 100 + tauler[t1, t2].2 | t1, t2 in TEAM where t1 <
t2], input_order, indomain_min)
]) minimize sum(t1 in TEAM, t2 in TEAM)(spectators_left_out[t1, t2]);
```

1.2.6. Restriccions implicades i simetries

Com a restriccions implicades tenim que:

1. Un enfrontament equip-equip només es pot produir una vegada en un dia i un estadi concret.
2. El fet d'haver d'omplir totes les caselles implica que els equips jugaran tots entre sí forçosament.

Les simetries trobades consisteixen en els dos triangles que resulten de separar la matriu per la seva diagonal. S'han solucionat amb les primeres constraints de la Secció 1.2.3..

1.3. *Viewpoint* 3

No cal entrar en profunditat en aquest *viewpoint* doncs utilitza les mateixes constraints que en el segon *viewpoint* i els mateixos solucionadors, però amb la particularitat de que aquest cop en comptes d'una matriu que guarda tuples, treballarem amb dues matrius equip-equip on una guardarà l'estadi i la altra el dia:

- `array[TEAM, TEAM] of var 0..ndays: dies`
- `array[TEAM, TEAM] of var 0..nstadiums: estadis`

La resta del *viewpoint* romandrà igual adaptat a aquest canvi.

1.4. Comparació dels models

Els nostres models ofereixen un millor rendiment quan s'intenta minimitzar el nombre de seguidors que quan s'intenta minimitzar la distància a recórrer dels equips. Així mateix, en les versions "mixtes" de les solucions, el seu rendiment també és lleugerament inferior.

Tanmateix, es pot veure clarament que la versió 1 amb els conjunts no dona un rendiment òptim i escala molt malament. Per aquest motiu, vam decidir implementar un model amb un viewpoint totalment diferent, amb tuples. En aquest segon cas, hem comprovat empíricament que és més òptim treballar amb un array de tuples binàries que no pas amb dues arrays. Així doncs, a la carpeta "`outputs`" del repositori s'hi poden trobar alguns dels outputs que hem generat en funció dels diferents valors i el temps que han tardat a certificar-los (si s'escau).

Es pot veure que el model que dona més bons resultats és el v2, seguit pròximament de l'v3 i on l'v1 queda molt lluny.

Després d'experimentar amb les diferents estratègies de cerca, hem trobat que la següent seqüència

```
seq_search([
    int_search([team_distance[t] | t in TEAM], input_order, indomain_min),
    int_search([spectators_left_out[t1, t2] | t1, t2 in TEAM where t1 < t2], input_order,
indomain_min),
    int_search([tauler[t1,t2].1 * 100 + tauler[t1,t2].2 | t1,t2 in TEAM where t1 < t2],
input_order, indomain_min)
])
```

és la que funciona millor. El que fa és:

1. Buscar el valor mínim possible de les distàncies entre equips.
2. Buscar el valor mínim d'espectadors fora els estadis.
3. Amb aquestes restriccions, llavors emplena el tauler de forma òptima.

Malgrat això, els models són incapaços de certificar les solucions que troben pels inputs `t3` i `t4`, ja siguin amb estadis fixats o sense. Tanmateix, sí que ha estat possible certificar que `t0` és insatisfactible, i les solucions de `t1` i `t2` amb i sense partits prefixats.

Instància	V1 (max seg.)	V2 (min dist.)	V2 (max seg.)	V3 (max seg.)
`t0`	1m 19s UNSAT	15s UNSAT	15s UNSAT	21s UNSAT
`t1`	13s NO CERT	19s NO CERT	13s CERT	37s CERT
`t1_fixe`	1.5s CERT	0.6s CERT	0.7s CERT	2.9s CERT
`t2`	-	-	54m 27s CERT	-
`t2_fixe`	-	48m 24s NO CERT	1m 58s CERT	1h 2m CERT
`t3`	-	-	32m 13s NO CERT	-
`t3_fixe`	-	-	29m 11s NO CERT	-
`t4`	-	-	53m 38s NO CERT	-
`t4_fixe`	-	-	54m 11s NO CERT	-

Taula 1: Resultats dels diferents models envers les instàncies proporcionades.[2] El solver utilitzat ha estat "OR Tools CP-SAT 9.11.4210[3] amb -O3, 16 threads[4] i llavor aleatòria "123123".

[2]Els resultats amb més detall, l'output corresponent a la solució, els temps exactes per a cada solució trobada i el nombre de propagacions, variables i reinicis es poden trobar a la carpeta `outputs` del repositori, dins la carpeta de `Minizinc`.

[3]Els rendiment dels altres solvers era molt més baix que l'OR Tools, per aquest motiu no s'han inclòs en el benchmark.

[4]L'equip utilitzat per a les proves és un Ryzen 7 5800X (8 cores, 16 threads) amb 32GB de RAM.

2. Part SAT

2.1. Explicació del model

2.1.1. Viewpoint

El model consisteix simplement en una matriu de dimensions `nombre_files x nombre_columnes` variables. Si hi ha una mina en la posició `[i,j]`, la variable serà certa. Altrament, serà falsa.

2.1.2. Restriccions bàsiques

1. Si ens indiquen el nombre de mines que ha de tenir el tauler, hem de posar un *exists K*:

```
if (nmines != -1) {
    e.addEK(tauler.flatten.toList, nmines)
}
```

2. Per a cadascuna de les caselles de la matriu llegida farem el següent:
   - Si és un "-", no cal fer res.
   - Si és una X, cal indicar que en aquella casella no hi ha una mina.
   - Si és un número, cal:
     - Indicar que en aquella casella no hi ha una mina.
     - Indicar que a les caselles veïnes hi ha d'haver *número* mines.

```
matrix.zipWithIndex.foreach {
  case (arr, i) => arr.zipWithIndex.foreach {
      case (variable, j) =>
          variable match {
              case "-" =>
              case "X" => e.addClause(-tauler(i)(j) :: List())
              case _ => addSurroundingEK(variable.toInt, i, j)
          }
  }
}
```

   - En funció del número, utilitzarem una restricció o una altra per a generar menys variables i reduir el nombre de constraints necessaris.
     - Si hi ha 0 veïns, totes les cel·les veïnes han de ser falses forçosament.
     - Si hi ha 1 veí, utilitzarem l'*exists one*.
     - Altrament, utilitzarem l'*exists K*, que ens generarà variables addicionals.

```
private val directions = List(
        (-1, -1), (-1, 0), (-1, 1),  // Top row: left, center, right
        (0, -1), (0, 1),             // Middle row: left, right
        (1, -1), (1, 0), (1, 1)      // Bottom row: left, center, right
    )

private def addSurroundingEK(neighbours: Int, x: Int, y: Int): Unit = {
    val veinsPossibles =
        for {
            (di, dj) <- directions
            ni = x + di
            nj = y + dj
            if ni >= 0 && ni < n && nj >= 0 && nj < m
```

```
        } yield (ni, nj)

    if (neighbours == 0) {
        // a les caselles surrounding segur que no hi ha una mina
        veinsPossibles.foreach(v => e.addClause(-tauler(v._1)(v._2) :: List()))
    }
    else if (neighbours == 1) {
        //e.addEOLog(veinsPossibles.map(v => tauler(v._1)(v._2)))
        e.addEOQuad(veinsPossibles.map(v => tauler(v._1)(v._2)))
    }
    else {
        e.addEK(veinsPossibles.map(v => tauler(v._1)(v._2)), K = neighbours)
    }
    e.addClause(-tauler(x)(y) :: List()) // en la casella de la restricció segur
que no hi ha una mina
}
```

2.1.3. Restriccions implicades

Aquest viewpoint implica que en una posició únicament hi podrà haver 0 o 1 mines, que és exactament el que ens interessa pel problema. A més a més, és molt senzill representar les restriccions de veïnatge amb la matriu auxiliar `directions`.

No hi ha cap altra restricció implicada.

2.1.4. Trencament de simetries

No s'observa cap simetria.

2.2. *Cardinality Constraints*

2.2.1. Codificació logarítmica de l'*at-most-one*

Per a codificar l'*at-most-one* d'una forma similar a l'explicació de les transparències de teoria, hem proposat la següent implementació:

```
def addAMOLog(l: List[Int]): Unit = {
    val twoPow = (Math.log(l.length) / Math.log(2)).ceil.toInt
    val newVariables = newVarArray(twoPow)

    def addClauses(variable: Int, numDec: Int): Unit = {
        newVariables.zipWithIndex.foreach {
            case (v, i) =>
                val bitValue = ((numDec >> i) & 1) * 2 - 1
                addClause(-variable :: (bitValue * v) :: Nil)
        }
    }

    l.zipWithIndex.foreach {
        case (value, index) => addClauses(value, index)
    }
}
```

La idea és que ens cal generar tantes variables com el $\log_2$ de la llargada de la llista. A continuació, ens cal generar, per a cada nova variable, les diferents combinacions de la representació binària (és a dir, $l[0] \to \overline{y_0}$ si en la representació binària equival a 0 o $l[0] \to y_0$ altrament). Per aquest motiu, utilitzem operacions a nivell de bit:

1. Agafem l'índex (`numDec`) i el desplacem cap a la dreta (*right shift*) i vegades. D'aquesta forma, ens queda l'i-èssim bit a la posició menys significant.
2. Fem una `and` amb 1 per a quedar-nos amb el valor de la i-èssima posició.
3. Convertim el 0 a un -1 i l'1 a un $+1$ per a facilitar l'afegiment de la clàusula.
4. Codifiquem l'expressió $l[0] \rightarrow \overline{y_0}$ cap a $\left(\overline{l[0]} \vee \overline{y_0}\right)$ i la inserim a la llista de clàusules de la codificació SAT. El signe negatiu ens permet realitzar la multiplicació `(bitValue * v)` que redueix el codi d'una forma elegant i equival al $\overline{y_0}$ o y_0.

Finalment, apliquem aquesta codificació logarítmica a l'*exactly-one* de la següent forma:

```
//Adds the encoding of the exactly-one using the logarithmic at-most-one
def addEOLog(l: List[Int]): Unit = {
    addAMOLog(l)
    addALO(l)
}
```

2.2.2. Comparació de configuracions

La diferència entre la configuració logarítmica i la quadràtica a l'hora de resoldre el problema del buscamines només té efecte en les caselles en què hi ha un "1" com a pista, doncs utilitzarem la funció *exactly one* per a les cel·les veïnes d'aquesta. Per la resta de caselles hem fet servir l'encoding proporcionat del sorter, que no utilitza ni la crida quadràtica ni la logarítmica. Així doncs, l'impacte que tindrà escollir una configuració o una altra serà mínim i es veurà reflectit per la mida del tauler a resoldre.

La hipòtesi que tenim és que per a tauler molt grans on el nombre de variables és elevat, utilitzar codificacions logarítmiques empitjorarà el rendiment. Això és degut a que l'*exactly one* s'aplica únicament a les caselles veïnes. Això implica afegir 2 o 3 variables per a cada pista que sigui un "1" (com a màxim $\log_2(8) = 3$, el mínim serien les cantonades i implicarien $\lceil \log_2(3) \rceil = 2$). Sembla lògic pensar que afegir $8^2 = 64$ clàusules binàries en el pitjor cas pugui ser millor que afegir $3n$ variables per a cada pista.

Com es pot veure a la Taula 2 de la Secció 2.3., sembla que aquesta hipòtesi podria considerar-se versemblant.

2.3. Resultats

2.3.1. Joc de proves inicial

Els resultats que a continuació es mostren resulten de les mitjanes de 5 execucions de cada instància per cada codificació. Això es deu a que el temps d'execució és tan petit que considerem que no podem fiarnos d'una sola mostra, doncs qualsevol petita pertorbació en l'equip podria condicionar els resultats.

Instància	Codificacions quadràtiques (s)	Codificacions logarítmiques (s)
37	0.002767611	0.002311294
50	0.006838272	0.006861232
70	0.008372983	0.005464091
86	0.019431836	0.017502934
90	0.125197325	0.154232472
140	0.016103461	0.026312013
273	0.007012866	0.008931608
329	0.020140511	0.026216714
350	0.017863223	0.018745854

Taula 2: Resultats de les instàncies inicials del joc de proves per les dues codificacions treballades.

Pensem que els resultats assoleixen l'objectiu, doncs el temps de resolució del sudoku més "difícil" és pràcticament immediat.

Com es pot observar a la Taula 2, per les instàncies més petites i senzilles, les codificacions logarítmiques son lleugerament més eficients, mentre que les quadràtiques ho son quan la complexitat augmenta.

Val a destacar que en la instància 90 el cost temporal es dispara. Això es deu a que té la característica especial de tenir un nombre de mines determinat, que afegeix una restricció *exactly k*.

A més té "X" inicials, és a dir, llocs on no hi pot haver una mina, però que tampoc donen informació de les seves caselles contigües. Tot i semblar que aquests dos aspectes son positius pel càlcul de la solució, afegeixen moltes codificacions que entorpeixen l'execució.

És per això que ens hem decidit a crear nosaltres mateixos algunes instàncies més grans.

2.3.2. Proves amb instàncies més grans

En vist l'èxit de l'apartat anterior, hem decidit crear instàncies molt més grans. Hem utilitzat un script de Python (Secció 1.).

L'script treballa amb 3 paràmetres dels quals 2 son les dimensions, i el tercer les bombes que es vulguin crear.

S'ha generat un tauler de 400x400 sense nombre de bombes concret, i de mitjana resol amb 88.4345046 segons.

Observem que en estipular un nombre de bombes pel mateix tauler, el programa no resol, doncs confirmem així que la restricció més costosa es la *exactly k*, que a dimensions molt grans escala molt.

Tenim la hipòtesis que quantes més bombes, més escala, doncs hi ha més clàusules i per tant més restriccions a complir.

Annex

1. Script python per generar taulers de Minesweeper

```python
import random

def generate_minesweeper_board(rows, cols, num_mines):
    # Initialize an empty grid
    board = [['0' for _ in range(cols)] for _ in range(rows)]

    # Place mines randomly
    mines_placed = 0
    while mines_placed < num_mines:
        row = random.randint(0, rows - 1)
        col = random.randint(0, cols - 1)
        if board[row][col] != '-1':  # -1 represents a mine
            board[row][col] = '-1'
            mines_placed += 1

    # Calculate the numbers on the board based on surrounding mines
    for row in range(rows):
        for col in range(cols):
            if board[row][col] == '-1':
                continue  # Skip mines
            # Count mines in the neighboring cells
            mine_count = 0
            for dr in [-1, 0, 1]:
                for dc in [-1, 0, 1]:
                    nr, nc = row + dr, col + dc
                    if 0 <= nr < rows and 0 <= nc < cols and board[nr][nc] == '-1':
                        mine_count += 1
            # Set the number if there are any mines around
            board[row][col] = str(mine_count) if mine_count > 0 else '0'

    return board


def hide_board(board, rows, cols):
    # Copy the board for hiding the numbers
    hidden_board = [['-' for _ in range(cols)] for _ in range(rows)]

    # We will leave some numbers visible to make the board solvable
    for row in range(rows):
        for col in range(cols):
            # Leave numbers visible if they are important to solving
            if board[row][col] != '0' and board[row][col] != '-1':
                if random.random() < 0.3:  # 30% chance to reveal a clue
                    hidden_board[row][col] = board[row][col]

    return hidden_board


def print_board(board):
    for row in board:
        print(" ".join(row))
```

```
def write_board_to_file(filename, rows, cols, num_mines, board, print_mines):
    # Open the file for writing (this will overwrite any existing content)
    with open(filename, 'w') as file:
        # Write the first line with NUM_ROWS NUM_COLS -1
        file.write(f"{rows} {cols} {num_mines if print_mines else -1}\n")

        # Write the board matrix
        for row in board:
            file.write(" ".join(row) + "\n")


# Parameters for the board
rows = 200  # Number of rows
cols = 200  # Number of columns
num_mines = 10000  # Number of mines

# Generate the Minesweeper board
board = generate_minesweeper_board(rows, cols, num_mines)

# Hide the clues and bombs (while making the board solvable)
hidden_board = hide_board(board, rows, cols)

# Output the hidden board to the console (optional)
print_board(hidden_board)

# Write the board to the output.txt file
write_board_to_file("output.txt",    rows,    cols,    num_mines,    hidden_board,
print_mines=False)
```