tags: `Report`

# Real-Life Applications of Computational Algorithms - HW1

309551130 謝柏威

## File structure

```
.
├── code
│   ├── 16
│   ├── 25
│   ├── 81
│   ├── 9
│   ├── MiniSat_v1.14_linux
│   ├── main.cpp
│   └── makefile
└── report.pdf

1 directory, 8 files
```

# Implementation

- The program is implemented in modern c++, which will requires the compile flag with `std=c++20`.
- All the detail will be explained later.

## 1. global variable

- `N` stands for the sudoku size, and `n` is sqrt(N).
- I maintain three bitset `row, col and box`, which can be used to check if certain number is valid in a block in constant time.

```
int N, n;
std::bitset<MAX_BOARD_SIZE> row[MAX_BOARD_SIZE], col[MAX_BOARD_SIZE], box[MAX_BOARD_SIZE];
```

## 2. preprocess

1. I read all the digits from the input file, and get `N` and `n`.

2. After reading the input, I create the sudoku board called `board` and fill in with the input.

```
// input sudoku digits
auto fin = std::fstream(argv[1], std::ios::in);
auto tmp = std::vector<int>{};
for (int x; fin >> x; tmp.push_back(x));
fin.close();

// initial bit mask
N = sqrt(tmp.size());
n = sqrt(N);
for (int i = 0; i < N; i++) row[i].set(), col[i].set(), box[i].set();

// set mask && build sudoku board
auto board = std::vector(N, std::vector<int>(N));
for (int i = 0, idx = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        board[i][j] = tmp[idx++];
        if (board[i][j] != 0) set_mask(board[i][j], i, j, 0);
    }
}
```

3. I set the mask for `row, col, and box`. For example, if `board[1][2] == 3`, then `row[1][3]`, `col[2][3]` and `box[0][3]` will be set to `0`, meaning that in these places the digit `3` is not allowed.

```
inline int block_idx(int x, int y) {
    return n * (x / n) + (y / n);
}

inline void set_mask(int num, int x, int y, int mask) {
    row[x][num] = mask;
    col[y][num] = mask;
    box[block_idx(x, y)][num] = mask;
}
```

# 3. encode

1. `cnf` is where I store all the clauses.
2. naive encoding: for each row, column and box, each digits appear exactly once.

```
auto cnf = std::vector<std::vector<int>>{};

// generate clause for (row, col, box, single block)
auto cand = std::vector<int>{};
for (int i = 0; i < N; i++) {
    for (int num = 1; num <= N; num++) {
        // row
```

```
        for (int j = 0; j < N; j++) {
            cand.push_back(to_var(i, j, num));
        }
        to_clause(cnf, cand);

        // col
        for (int j = 0; j < N; j++) {
            cand.push_back(to_var(j, i, num));
        }
        to_clause(cnf, cand);

        // box
        for (int j = i / n * n; j < (i / n + 1) * n; j++) {
            for (int k = i % n * n; k < (i % n + 1) * n; k++) {
                cand.push_back(to_var(j, k, num));
            }
        }
        to_clause(cnf, cand);
    }
}
```

3. naive encoding: for each block in sudoku, each block can only be a number between 1~N.

```
// single block
for (int i = 0; i < N; i++) {
    for (int j = 0; j < N; j++) {
        for (int num = 1; num <= N; num++) {
            if (board[i][j] == 0 and is_valid(i, j, num)) {
                cand.push_back(to_var(i, j, num));
            } else if (board[i][j] != 0) {
                cnf.push_back({to_var(i, j, num) * (num == board[i][j] ? 1 : -1)});
            }
        }
        to_clause(cnf, cand);
    }
}
```

4. check if a digit is valid in certain block in O(1) time complexity.

```
inline int block_idx(int x, int y) {
    return n * (x / n) + (y / n);
}

inline bool is_valid(int x, int y, int guess) {
    return row[x][guess] & col[y][guess] & box[block_idx(x, y)][guess];
}
```

5. `to_var` function converts the index(row, column, digit) to cnf variable

```
// transfer index(r, c, d) to cnf variable
auto to_var (int r, int c, int d) {
    return (r * N + c) * N + d;
}
```

6. `to_clause` function do the exactly-once encoding

```
// exactly-once encoding (clauses, vector of cnf variables)
    int s = v.size();
    if (s == 0) return;

    // at least one
    cnf.push_back(v);

    // at most one
    for (int i = 0; i < s; i++) {
        for (int j = i + 1; j < s; j++) {
            cnf.push_back({-v[i], -v[j]});
        }
    }
    v.clear();
}
```

7. finally, after the encoding, output all the clauses to `sat.in` as the input of the mini-sat solver.

```
// encoding done (SAT input)
auto fout = std::fstream("sat.in", std::ios::out | std::ios::trunc);
fout << "p cnf " << pow(N, 3) << " " << cnf.size() << "\n";
for (auto& v : cnf) {
    fout << v << " 0\n";
}
fout.close();
```

# 4. decode

1. execution. ( `sat.out` is the ouput file for MiniSat)

```
system("./MiniSat_v1.14_linux sat.in sat.out");
```

2. decode from SAT to sudoku

```
bool decode (auto& board) {
    auto fin = std::fstream("sat.out", std::ios::in);
    auto buf = std::string{};
    std::getline(fin, buf);
    if (buf == "UNSAT") return false;
```

```
    std::getline(fin, buf);

    // decode variables back to sudoku
    auto ss = std::stringstream{buf};
    for (int x; ss >> x and x != 0; ) {
        if (x < 0) continue;
        auto [r, c, d] = to_idx(x);
        board[r][c] = d;
    }

    fin.close();
    return true;
}
```

3. convert the cnf variables back to (row, col, digit)

```
// transfer cnf variable to index(r, c, d)
auto to_idx (int x) {
    x -= 1;
    int rc = x / N;
    return std::tuple {rc / N, rc % N, x % N + 1};
}
```

# Results

Following code is my simple `makefile`. Simple type `make` will compile and run with the default settings.

```
CC = g++-10
ARGS = -std=c++20 -O2 -Wall -Wextra -Wshadow -Wpedantic

SRC = main.cpp
TAR = solver
SIZE = 16
ANS = answer.txt

run: solver
	./solver $(SIZE) $(ANS) MiniSat_v1.14_linux

solver: makefile $(SRC)
	$(CC) $(ARGS) $(SRC) -o $(TAR)

clean:
	rm -rf a.out sat.in sat.out $(ANS) $(TAR)
```

## 1. SIZE = 9 * 9

```
$ make SIZE=9                                                                50ms
./solver 9 answer.txt MiniSat_v1.14_linux
==================================[MINISAT]===================================
| Conflicts |     ORIGINAL     |              LEARNT              | Progress |
|           | Clauses Literals |   Limit Clauses Literals  Lit/Cl |          |
==============================================================================
|         0 |     871    18204 |     290       0        0     nan |  0.000 % |
==============================================================================
restarts              : 1
conflicts             : 0              (0 /sec)
decisions             : 1              (152 /sec)
propagations          : 729            (110925 /sec)
conflict literals     : 0              ( nan % deleted)
Memory used           : 1.81 MB
CPU time              : 0.006572 s

SATISFIABLE
```

## 2. SIZE = 25 * 25

```
$ make SIZE=25                                                                431ms
./solver 25 answer.txt MiniSat_v1.14_linux
==================================[MINISAT]===================================
| Conflicts |     ORIGINAL     |              LEARNT              | Progress |
|           | Clauses Literals |   Limit Clauses Literals  Lit/Cl |          |
==============================================================================
|         0 |   43545  1169340 |   14515       0        0     nan |  0.000 % |
==============================================================================
restarts              : 1
conflicts             : 78             (706 /sec)
decisions             : 607            (5495 /sec)
propagations          : 35215          (318803 /sec)
conflict literals     : 2554           (5.48 % deleted)
Memory used           : 9.04 MB
CPU time              : 0.11046 s

SATISFIABLE
```

## 3. SIZE = 81 * 81

```
$ make SIZE=81                                                               69s
./solver 81 answer.txt MiniSat_v1.14_linux
==================================[MINISAT]====================================
| Conflicts |      ORIGINAL      |              LEARNT              | Progress |
|           | Clauses  Literals |   Limit Clauses Literals  Lit/Cl |          |
===============================================================================
|         0 | 1060907 128399015 |   353635       0        0     nan |  0.000 % |
|       100 | 1060926 128399015 |   388998      66      712    10.8 | 96.316 % |
```

```
|       250 | 1060949 128399015 |  427898     163     1583     9.7 | 96.499 % |
|       475 | 1060981 128399015 |  470688     325     3266    10.0 | 96.591 % |
|       812 | 1061020 128399015 |  517757     587     8165    13.9 | 98.004 % |
|      1322 | 1061053 128399015 |  569532    1044    18054    17.3 | 98.344 % |
===============================================================================
restarts              : 6
conflicts             : 1611           (46 /sec)
decisions             : 10653          (305 /sec)
propagations          : 1478934        (42288 /sec)
conflict literals     : 26755          (12.21 % deleted)
Memory used           : 729.84 MB
CPU time              : 34.973 s

SATISFIABLE
```

# Discussion

- The result is actually surprising. I did not expect the naive encoding to have such fast performance. Even large sudoku like `81 x 81` takes only **a minute and a half** to encode + solve. If we look at the CPU time for MiniSat, it only takes **34s** to solve!
- After having a taste of what SAT solver could do, I'm looking forward to the later project whick we are going to implement a SAT solver ourself. Must be a lot of fun!