Programming Assignment #2 Report

0516007 高誌佑

1. Implementation Details

For the basic backtrack search, I implement it with a stack. Each node contains current variables with value assigned, domain of unassigned variable, and current number of mine for the conveneience of checking the constraint of total mine number. When a node was popped, do one of the following actions:

1. output the solution if all variables are assigned.
2. failure is detected if the domain of the current unassigned variable is empty, cut the node
3. generate the child nodes for every value in domain, update the domains after the assigning the value of the current variable, and insert child nodes into the stack.
   (I update the domains without doing forward checking. The empty domains of some variables will only be detected when child nodes is popped in the future.)

For the forward checking, do the consistency check before insert child nodes into the stack. The child nodes will not be inserted if some failures are detected. The consistenct check includes checking whether the domains of variables can satisfy all the hints and the number of total mine.

For MRV heuristic, when a node is popped, I use a linear search for finding an unassigned variable with the fewest legal value. Then, assign this variable first.

For Degree heuristic, before the backtrack search, I calculate the number of constraint of each variable and sort the variables according to the number. Since the number of constraint will not change during the process of search, I sort the variables in the beginning rather than use a linear search every time when a node is popped.

For LCV heuristic, for both values, I use the consistency check to calculate the sum of the size of domains of the affected variables. The values with less affected domains are tried first.

My code finds **all the solutions** rather than only a legal solution, since I noticed it after I have done all the experiments and the report. As a result, the experiment results and the analysis are based on the implementation of finding all solutions.

2. Experiments results, Observations, and Interpretations

Below are some settings in all experiments:

- Machine (CPU): AMD Ryzen 7-3700X 3.6GHz
- Operating System: Ubuntu 18.04
- Language: **C++**
- Compiler Version: gcc 7.5.0
- The execution time are the average time of five trial.

Execution Time and Expanded Nodes Comparison

This test compares the execution time and the number of expanded nodes with different board sizes, number of mines and hints. The following tables show the results. The red color means it is the fastest among basic search and three heuristics.

First, I test the board size 6x6 using the sample testcase1 given in the spec.

From the below table, we can see that the time and the expanded nodes are less than the basic backtrack search when we apply MRV and Degree heuristic. MRV tends to choose the variable with the fewest legal values, while Degree heuristic gives variables with more constraints higher priority. To a certain degree, more constraints means less selections, so both of them aim to handle variables with less selection first. In this way, it can further avoid some failures, and that is why the time and the number of expanded nodes are less.

As for LCV, it uses the consistency check to decide which value should be tried first. However, the execution time is longer than the basic search. I guess the reason is that the overhead is heavy and that I try value 0 first by default. Value 0 means the variable is not a mine, and it usually affects less domains of other variables in the game. Thus, LCV is hard to improve the performance. I also change my code that value 1 is tried first, and the execution time of LCV decreases a little, but is still longer than that of basic search.

For the forward checking, it detects the failures earlier, so it helps to reduce the execution time and the number of expanded nodes. The basic search and three heuristic all becomes faster.

- Board Size: 6 x 6, 10 mines, 16 hints

Algorithm	Time	Expanded Nodes
Basic	2.07 ms	156
MRV	1.08 ms	72
Degree	1.91 ms	139
LCV	2.61 ms	153
Basic + forward checking	1.85 ms	125
MRV + forward checking	0.94 ms	63
Degree + forward checking	1.22 ms	85
LCV + forward checking	2.49 ms	123

I also do the experiments with board size 9x9 (the testcase is randomly generated).

The results is similar to 6x6 board. The difference is that Degree heuristic performs better than MRV here. I think which one has better performance is case by case.

In addition, I experiment with more mines and hints. As the number of mines and hints grow, the number of solutions decreases. Therefore, the initial domains of all variables are limited, so failures are usually detected very early. It results in less execution time and expanded nodes.

- Board Size: 9 x 9, 15 mines, 24 hints

Algorithm	Time	Expanded Nodes
Basic	360 ms	43649
MRV	33 ms	2439
Degree	30 ms	1684
LCV	422 ms	43620
Basic + forward checking	126 ms	12319
MRV + forward checking	20 ms	1215
Degree + forward checking	17 ms	1012
LCV + forward checking	148 ms	12268

- Board Size: 9 x 9, 22 mines, 36 hints

Algorithm	Time	Expanded Nodes
Basic	3.56 ms	196
MRV	0.87 ms	45
Degree	0.90 ms	49
LCV	4.19 ms	186
Basic + forward checking	2.34 ms	116
MRV + forward checking	0.86 ms	45
Degree + forward checking	0.87 ms	45
LCV + forward checking	2.78 ms	107

In the last, I do the experiments with board size 12x12 (the testcase is randomly generated).

The execution time and the number of expanded nodes become larger. The basic search and LCV even cannot finish in one minute. However, it still finishes quickly when there are many mines and hints. Here, we can easily realize the importance of a good heuristic in searching.

- Board Size: 12 x 12, 30 mines, 48 hints

Algorithm	Time	Expanded Nodes
Basic	> 60 s	-
MRV	5.82 s	428315
Degree	3.27 s	179619
LCV	> 60 s	-
Basic + forward checking	> 60 s	-
MRV + forward checking	1.93 s	153907
Degree + forward checking	0.81 s	56108
LCV + forward checking	> 60 s	-

- Board Size: 12 x 12, 40 mines, 64 hints

Algorithm	Time	Expanded Nodes
Basic	117 ms	9585
MRV	10 ms	413
Degree	25 s	1812
LCV	131 ms	9453
Basic + forward checking	62 ms	3959
MRV + forward checking	8 ms	323
Degree + forward checking	13 ms	551
LCV + forward checking	67 ms	3894

3. Things I have learned

It is the first time that I implement these three kinds of heuristics. It is very interesting to combine the searching with different heuristics. A good heuristic can enhance the performance very much. I indeed benefit a lot from thinking why these heuistics works. In addition, I learn the concept of forward checking. I will think about these concepts when I implement searching next time.

4. Remaining Questions and Future Investigations

In this programming assignment, the performance of LCV seems not good. I am wondering in which case that LCV will speedup a lot. In the future, I will also attemp to find a better heuristic function to further improve the performance.

Appendix: Source code

- Compile Command: g++ -g -Wall -o search main.cpp
- main.cpp

```
#include <bits/stdc++.h>
using namespace std;

const int MAX_SIZE = 100;
int board[MAX_SIZE][MAX_SIZE];
char c_board[MAX_SIZE][MAX_SIZE];    // for output solution
int constraints[MAX_SIZE][MAX_SIZE]; // for degree heuristic

const int dx[8] = {-1, -1, -1, 0, 0, 1, 1, 1};
const int dy[8] = {-1, 0, 1, -1, 1, -1, 0, 1};

int board_size_x, board_size_y, mine_total, solution_num = 0;
bool forward_check;
int heuristic_type;

struct Node {
   vector<vector<int>> assignments;  // variables with value assigned from root to this node
   vector<vector<int>> domains;      // domains of unassigned variables
   vector<pair<int, int>> unassigned;// coordinate of unassigned variables
   int current_mine_num; // the number of mines from root to this node

   Node () {
      assignments.resize(board_size_x, vector<int>(board_size_y, 0));
      domains.resize(board_size_x, vector<int>(board_size_y, -1));
      unassigned.clear();
      current_mine_num = 0;
   }

   Node (const vector<vector<int>> &assignments, const vector<vector<int>> &domains, \
         const vector<pair<int, int>> &unassigned, int current_mine_num)
      : assignments(assignments), domains(domains), unassigned(unassigned),\
        current_mine_num(current_mine_num) {}
};

// sort according to constraints in Degree heuristic
struct Degree_cmp {
   bool operator() (pair<int, int> a, pair<int, int> b) {
      return constraints[a.first][a.second] > constraints[b.first][b.second];
   }
};
```

```
bool is_outside(int x, int y) {
    return x < 0 || x >= board_size_x || y < 0 || y >= board_size_y;
}

void output(const vector<vector<int>> &assignments) {
    int mine_num = 0;
    for (int i = 0; i < board_size_x; i++) {
        for (int j = 0; j < board_size_y; j++) {
            if (board[i][j] == -1) {
                if (assignments[i][j] == 0)   c_board[i][j] = '-';
                if (assignments[i][j] == 1)   c_board[i][j] = '*', mine_num++;
            }
            else {
                c_board[i][j] = char('0' + board[i][j]);
            }
        }
    }
    if (mine_num != mine_total) return ;

    ++solution_num;
    // cout << "Solution " << solution_num << ":\n";
    // for (int i = 0; i < board_size_x; i++) {
    //     for (int j = 0; j < board_size_y; j++) {
    //         cout << c_board[i][j] << " ";
    //     }
    //     cout << "\n";
    // }
    // cout << "======================\n";
}

void init_root(Node &node) {
    // init the value assignments of variables
    for (int x = 0; x < board_size_x; x++) {
        for (int y = 0; y < board_size_y; y++) {
            if (board[x][y] == -1) {
                node.assignments[x][y] = -1;
                node.unassigned.push_back(make_pair(x, y));
            }
        }
    }

    // init the domains (and constraints) of variables
    memset(constraints, 0, sizeof(constraints));
    for (int x = 0; x < board_size_x; x++) {
        for (int y = 0; y < board_size_y; y++) {
```

```
            if (board[x][y] != -1) continue;

            int current_domain = 0b11;
            for (int i = 0; i < 8; i++) {
                int center_x = x + dx[i];
                int center_y = y + dy[i];
                if (is_outside(center_x, center_y))   continue;
                if (board[center_x][center_y] == -1)  continue;

                constraints[x][y]++;

                int mine_num = 0;
                int space_num = 0;
                for (int j = 0; j < 8; j++) {
                    int outer_x = center_x + dx[j];
                    int outer_y = center_y + dy[j];
                    if (is_outside(outer_x, outer_y))   continue;

                    if (board[outer_x][outer_y] == -1) {
                        if (node.assignments[outer_x][outer_y] == 1)     mine_num++;
                        if (node.assignments[outer_x][outer_y] == -1)    space_num++;
                    }
                }

                int mine_need = board[center_x][center_y] - mine_num;
                if (mine_need == 0)              current_domain &= 0b01;
                else if (mine_need == space_num) current_domain &= 0b10;
                else if (mine_need < space_num)  current_domain &= 0b11;
                else if (mine_need > space_num)  current_domain &= 0b00;

                if (current_domain == 0b00) break;
            }
            node.domains[x][y] = current_domain;
        }
    }

    if (heuristic_type == 2)    sort(node.unassigned.begin(), node.unassigned.end(), Degree_cmp());
}

void backtrack_search(const Node &root) {
    int num_expand = 0;
    stack<Node> frontier;
    frontier.push(root);

    while (!frontier.empty()) {
```

```
Node node = frontier.top(); frontier.pop();

// successfully find a solution when all variables are assigned
if (node.unassigned.empty()) {
    output(node.assignments);
    continue;
}

num_expand++;
pair<int, int> variable = node.unassigned.front();

int x = variable.first, y = variable.second;
int current_domain = node.domains[x][y];

// if the domain is empty then cut the node
if (current_domain == 0)  continue;

vector<int> value_bit{0, 1};

// LCV (only when domain is still {0, 1})
if (heuristic_type == 3 && current_domain == 0b11) {
    int domain_affect[2] = {0, 0};
    for (int k : value_bit) {
        vector<vector<int>> assignments = node.assignments;
        vector<vector<int>> domains = node.domains;
        vector<pair<int, int>> unassigned;
        unassigned.assign(node.unassigned.begin() + 1, node.unassigned.end());
        assignments[x][y] = k;

        bool not_satisfied = false; // forward checking

        for (int i = 0; i < 8; i++) {
            int center_x = x + dx[i];
            int center_y = y + dy[i];
            if (is_outside(center_x, center_y))   continue;
            if (board[center_x][center_y] == -1)  continue;

            int mine_num = 0, space_num = 0;
            for (int j = 0; j < 8; j++) {
                int outer_x = center_x + dx[j];
                int outer_y = center_y + dy[j];
                if (is_outside(outer_x, outer_y))   continue;

                if (board[outer_x][outer_y] == -1) {
                    if (assignments[outer_x][outer_y] == 1)    mine_num++;
```

```
                if (assignments[outer_x][outer_y] == -1)   space_num++;
            }
        }
        int mine_need = board[center_x][center_y] - mine_num;
        int update_domain;
        if (mine_need == 0)             update_domain = 0b01;
        else if (mine_need == space_num) update_domain = 0b10;
        else if (mine_need < space_num)  update_domain = 0b11;
        else if (mine_need > space_num)  update_domain = 0b00;

        for (int j = 0; j < 8; j++) {
            int outer_x = center_x + dx[j];
            int outer_y = center_y + dy[j];
            if (is_outside(outer_x, outer_y))   continue;

            if (board[outer_x][outer_y] == -1 && assignments[outer_x][outer_y] == -1) {
                domain_affect[k] += __builtin_popcount(domains[outer_x][outer_y]);
                domains[outer_x][outer_y] &= update_domain;
                domain_affect[k] -= __builtin_popcount(domains[outer_x][outer_y]);
                if (domains[outer_x][outer_y] == 0) {
                    not_satisfied = true;
                }
            }
        }
    }
    if (not_satisfied) domain_affect[k] = INT_MAX;
  }
  // if value 1 is better, try value 1 first
  if (domain_affect[0] > domain_affect[1]) {
    swap(value_bit[0], value_bit[1]);
  }
}

for (int k : value_bit) {
  if ((current_domain & (1 << k)) == 0)   continue;

  vector<vector<int>> assignments = node.assignments;
  vector<vector<int>> domains = node.domains;
  vector<pair<int, int>> unassigned;
  unassigned.assign(node.unassigned.begin() + 1, node.unassigned.end());
  int current_mine_num = node.current_mine_num + k;
  assignments[x][y] = k;

  bool not_satisfied = false; // forward checking
```

```
// update the domains of other variables
for (int i = 0; i < 8; i++) {
    int center_x = x + dx[i];
    int center_y = y + dy[i];
    if (is_outside(center_x, center_y))   continue;
    if (board[center_x][center_y] == -1)  continue;

    int mine_num = 0, space_num = 0;
    for (int j = 0; j < 8; j++) {
        int outer_x = center_x + dx[j];
        int outer_y = center_y + dy[j];
        if (is_outside(outer_x, outer_y))   continue;

        if (board[outer_x][outer_y] == -1) {
            if (assignments[outer_x][outer_y] == 1)    mine_num++;
            if (assignments[outer_x][outer_y] == -1)   space_num++;
        }
    }
    int mine_need = board[center_x][center_y] - mine_num;
    int update_domain;
    if (mine_need == 0)              update_domain = 0b01;
    else if (mine_need == space_num) update_domain = 0b10;
    else if (mine_need < space_num)  update_domain = 0b11;
    else if (mine_need > space_num)  update_domain = 0b00;

    for (int j = 0; j < 8; j++) {
        int outer_x = center_x + dx[j];
        int outer_y = center_y + dy[j];
        if (is_outside(outer_x, outer_y))   continue;

        if (board[outer_x][outer_y] == -1 && assignments[outer_x][outer_y] == -1) {
            domains[outer_x][outer_y] &= update_domain;
            if (forward_check && domains[outer_x][outer_y] == 0) {
                not_satisfied = true;
            }
        }
    }
}

// return earlier if constraints will not be satisfied
if (forward_check) {
    if (not_satisfied)  continue;
    int low_bound = 0, upp_bound = 0;
    for (const auto &variable : unassigned) {
        int vx = variable.first, vy = variable.second;
```

```
                if (domains[vx][vy] & 0b10)  upp_bound++;
                if (domains[vx][vy] == 0b10) low_bound++;
            }
            // check constraint of total mine number
            if ((low_bound + current_mine_num > mine_total) ||
                (upp_bound + current_mine_num < mine_total)) continue;
        }

        // MRV
        if (heuristic_type == 1) {
            int min_legal_value = 3, index = 0, len = unassigned.size();
            for (int i = 0; i < len; i++) {
                int vx = unassigned[i].first, vy = unassigned[i].second;
                int bits = __builtin_popcount(domains[vx][vy]);
                if (bits < min_legal_value) {
                    min_legal_value = bits;
                    index = i;
                }
            }
            if (index != 0) swap(unassigned[0], unassigned[index]);
        }

        frontier.push(Node{assignments, domains, unassigned, current_mine_num});
      }
    }

    cout << "Number of solutions: " << solution_num << "\n";
    cout << "Number of expanded nodes: " << num_expand << "\n";
}

int main(int argc, char **argv) {
    if (argc > 1) forward_check = (atoi(argv[1]) == 1);
    if (argc > 2) heuristic_type = atoi(argv[2]);
    else          heuristic_type = 0;
    cout << "forward_check: " << forward_check << " heuristic: " << heuristic_type << "\n";

    cin >> board_size_x >> board_size_y >> mine_total;
    Node root;
    for (int i = 0; i < board_size_x; i++) {
        for (int j = 0; j < board_size_y; j++) {
            cin >> board[i][j];
        }
    }
    init_root(root);
```

```
    auto t1 = std::chrono::high_resolution_clock::now();
    backtrack_search(root);
    auto t2 = std::chrono::high_resolution_clock::now();
    auto duration = std::chrono::duration_cast<std::chrono::microseconds>( t2 - t1 ).count();
    cout << float(duration) / 1000.0 << " ms\n";
    return 0;
}
```