School of Data and Computer Science, Sun Yat-sen University

E04 Futoshiki Puzzle (Forward Checking)

18364066 Yanzuo Lu

September 23, 2020

Contents

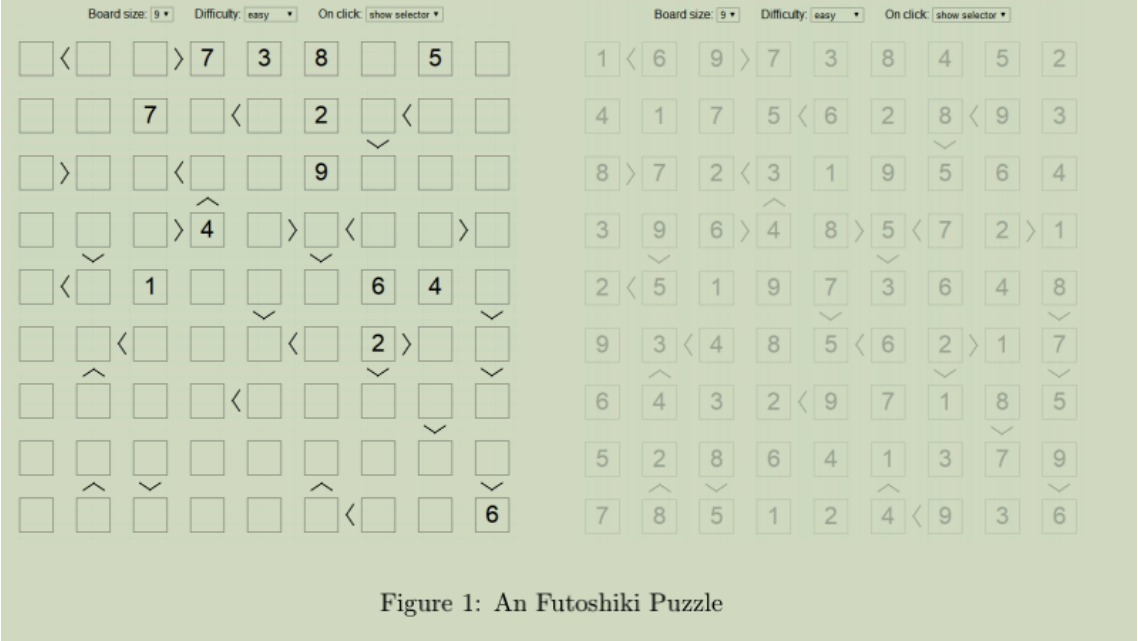

Figure 1: An Futoshiki Puzzle

Figure 1: Futoshiki

1 Futoshiki

Futoshiki is a board-based puzzle game, also known under the name Unequal. It is playable on a square board having a given fixed size (4×4 for example).

The purpose of the game is to discover the digits hidden inside the board' s cells; each cell is filled with a digit between 1 and the board' s size. On each row and column each digit appears exactly once; therefore, when revealed, the digits of the board form a so-called Latin square.

At the beginning of the game some digits might be revealed. The board might also contain some inequalities between the board cells; these inequalities must be respected and can be used as clues in order to discover the remaining hidden digits.

Each puzzle is guaranteed to have a solution and only one.

You can play this game online: http://www.futoshiki.org/.

2 Tasks

1. Please solve the above Futoshiki puzzle (Figure 1) with forward checking algorithm.
2. Write the related codes and take a screenshot of the running results in the file named E04 YourNumber.pdf, and send it to ai 2020@foxmail.com.

3 Codes

```
#include <iostream>
#include <vector>
#include <set>

using namespace std;

class FutoshikiPuzzle
{
public:
    vector<vector<int>> maps;
    vector<pair<pair<int, int>, pair<int, int>>> \
        less_constraints;
    int nRow, nColumn, nConstraint;

    bool assigned[9][9] = { false };
    vector<vector<set<int>>> current_domain;

    void addConstraints(int x, int y, int x1, int y1)
    {
        less_constraints.push_back({ {x, y}, {x1, y1} });
    }

    void initial()
    {
        //初始地图
        maps = { {0, 0, 0, 7, 3, 8, 0, 5, 0},
                 {0, 0, 7, 0, 0, 2, 0, 0, 0},
                 {0, 0, 0, 0, 0, 9, 0, 0, 0},
                 {0, 0, 0, 4, 0, 0, 0, 0, 0},
                 {0, 0, 1, 0, 0, 0, 6, 4, 0},
                 {0, 0, 0, 0, 0, 0, 2, 0, 0},
                 {0, 0, 0, 0, 0, 0, 0, 0, 0},
                 {0, 0, 0, 0, 0, 0, 0, 0, 0},
                 {0, 0, 0, 0, 0, 0, 0, 0, 6} };
        nRow = maps.size();
        nColumn = maps[0].size();

        //添加限制
        addConstraints(0, 0, 0, 1);
        addConstraints(0, 3, 0, 2);
        addConstraints(1, 3, 1, 4);
        addConstraints(1, 6, 1, 7);
        addConstraints(2, 6, 1, 6);
        addConstraints(2, 1, 2, 0);
        addConstraints(2, 2, 2, 3);
        addConstraints(2, 3, 3, 3);
```

```
addConstraints(3, 3, 3, 2);
addConstraints(3, 5, 3, 4);
addConstraints(3, 5, 3, 6);
addConstraints(3, 8, 3, 7);
addConstraints(4, 1, 3, 1);
addConstraints(4, 5, 3, 5);
addConstraints(4, 0, 4, 1);
addConstraints(5, 4, 4, 4);
addConstraints(5, 8, 4, 8);
addConstraints(5, 1, 5, 2);
addConstraints(5, 4, 5, 5);
addConstraints(5, 7, 5, 6);
addConstraints(5, 1, 6, 1);
addConstraints(6, 6, 5, 6);
addConstraints(6, 8, 5, 8);
addConstraints(6, 3, 6, 4);
addConstraints(7, 7, 6, 7);
addConstraints(7, 1, 8, 1);
addConstraints(8, 2, 7, 2);
addConstraints(7, 5, 8, 5);
addConstraints(8, 8, 7, 8);
addConstraints(8, 5, 8, 6);

//表示第x行中某个数字是否存在
int Count_RowNumbers[9][10] = { 0 };
//表示第y列某个数字是否存在
int Count_ColumnNumbers[9][10] = { 0 };

current_domain.resize(9);
for (int i = 0; i < 9; i++)
{
    current_domain[i].resize(9);
    for (int j = 0; j < 9; j++)
    {
        if (maps[i][j] != 0)
        {
            Count_RowNumbers[i][maps[i][j]]++;
            Count_ColumnNumbers[j][maps[i][j]]++;
            assigned[i][j] = true;
        }
    }
}

for (int i = 0; i < 9; i++)
{
    for (int j = 0; j < 9; j++)
    {
        for (int k = 1; k < 10; k++)
        {
```

```
                if (Count_RowNumbers[i][k] == 0 && \
                    Count_ColumnNumbers[j][k] == 0)
                {
                    current_domain[i][j].insert(k);
                }
            }
        }
    }

    nConstraint = less_constraints.size();
    for (int c = 0; c < nConstraint; c++)
    {
        int x = less_constraints[c].first.first;
        int y = less_constraints[c].first.second;
        int x1 = less_constraints[c].second.first;
        int y1 = less_constraints[c].second.second;
        if (assigned[x][y] && !assigned[x1][y1])
        {
            int data = maps[x][y];
            for (int k = 1; k < data; k++)
            {
                if (current_domain[x1][y1].find(k) != \
                    current_domain[x1][y1].end())
                {
                    current_domain[x1][y1].erase(k);
                }
            }
        }
        if (assigned[x1][y1] && !assigned[x][y])
        {
            int data = maps[x1][y1];
            for (int k = data + 1; k < 10; k++)
            {
                if (current_domain[x][y].find(k) != \
                    current_domain[x][y].end())
                {
                    current_domain[x][y].erase(k);
                }
            }
        }
    }

}

//显示图片
void show()
{
    for (int i = 0; i < nRow; i++)
    {
```

```
        for (int j = 0; j < nColumn; j++)
        {
            cout << maps[i][j] << " ";
        }
        cout << endl;
    }
    cout << "==========" << endl;
}

bool AllAssigned()
{
    for (int i = 0; i < 9; i++)
    {
        for (int j = 0; j < 9; j++)
        {
            if (!assigned[i][j]) return false;
        }
    }
    return true;
}

pair<int, int> PickAnUnassignedVariable()
{
    // Minimum Remaining Values Heuristics(MRV)
    int x = -1, y = -1;
    int minimum = 10;
    for (int i = 0; i < 9; i++)
    {
        for (int j = 0; j < 9; j++)
        {
            if (assigned[i][j]) continue;
            if (current_domain[i][j].size() < minimum)
            {
                x = i;
                y = j;
                minimum = current_domain[i][j].size();
            }
        }
    }
    return make_pair(x, y);
}

bool ForwardChecking(int level)
{
    pair<int, int> next_v = PickAnUnassignedVariable();
    int x = next_v.first, y = next_v.second;

    assigned[x][y] = true;
    vector<int> members;
```

```
for (auto iter = current_domain[x][y].begin(); iter != \
    current_domain[x][y].end(); iter++)
    members.push_back(*iter);
current_domain[x][y].clear();

for (auto iter = members.begin(); iter != members.end(); iter++)
{
    maps[x][y] = *iter;
    if (AllAssigned()) return true;

    vector<vector<vector<int>>> prnued_values;
    prnued_values.resize(9);
    for (int i = 0; i < 9; i++) prnued_values[i].resize(9);

    bool empty_flag = false;
    for (int k = 0; k < 9; k++)
    {
        if (current_domain[x][k].find(maps[x][y]) != \
            current_domain[x][k].end())
        {
            current_domain[x][k].erase(maps[x][y]);
            prnued_values[x][k].push_back(maps[x][y]);
        }
        if (current_domain[k][y].find(maps[x][y]) != \
            current_domain[k][y].end())
        {
            current_domain[k][y].erase(maps[x][y]);
            prnued_values[k][y].push_back(maps[x][y]);
        }
        if ((!assigned[x][k] && current_domain[x][k].empty()) || \
            (!assigned[k][y] && current_domain[k][y].empty()))
        {
            empty_flag = true;
            break;
        }
    }

    if (!empty_flag)
    {
        for (int c = 0; c < nConstraint; c++)
        {
            int x1 = less_constraints[c].first.first;
            int y1 = less_constraints[c].first.second;
            int x2 = less_constraints[c].second.first;
            int y2 = less_constraints[c].second.second;

            if (x1 == x && y1 == y && !assigned[x2][y2])
            {
                for (int k = 1; k < maps[x][y]; k++)
```

```
                {
                    if (current_domain[x2][y2].find(k) != \
                        current_domain[x2][y2].end())
                    {
                        current_domain[x2][y2].erase(k);
                        prnued_values[x2][y2].push_back(k);
                    }
                }
                if (current_domain[x2][y2].empty())
                {
                    empty_flag = true;
                    break;
                }
            }
            if (x2 == x && y2 == y && !assigned[x1][y1])
            {
                for (int k = maps[x][y] + 1; k < 10; k++)
                {
                    if (current_domain[x1][y1].find(k) != \
                        current_domain[x1][y1].end())
                    {
                        current_domain[x1][y1].erase(k);
                        prnued_values[x1][y1].push_back(k);
                    }
                }
                if (current_domain[x1][y1].empty())
                {
                    empty_flag = true;
                    break;
                }
            }
        }
    }

    if (!empty_flag)
    {
        if (ForwardChecking(level + 1))
        {
            return true;
        }
    }

    for (int i = 0; i < 9; i++)
    {
        for (int j = 0; j < 9; j++)
        {
            int len = prnued_values[i][j].size();
            for (int k = 0; k < len; k++)
            {
```

```
                        current_domain[i][j].insert(prnued_values[i][j][k]);
                    }
                }
            }
        }
        maps[x][y] = 0;
        for (auto iter = members.begin(); iter != members.end(); iter++)
        {
            current_domain[x][y].insert(*iter);
        }
        assigned[x][y] = false;
        return false;
    }
};

int main()
{
    FutoshikiPuzzle* futoshikiPuzzle = new FutoshikiPuzzle();
    futoshikiPuzzle->initial();
    futoshikiPuzzle->show();
    futoshikiPuzzle->ForwardChecking(1);
    futoshikiPuzzle->show();
    delete futoshikiPuzzle;
    return 0;
}
```

4 Results

本次实验重在考查 CSP 问题中的 ForwardChecking 向前检测算法，核心思想是在每一层选择一个节点并放置一个数字时，对行、列和大小约束三个方面对地图上的所有节点进行剪枝，假设我在 (0,2) 位置放置一个数字”9”，那么数字所在的第零行和第二列都不可能再出现数字”9”，与此同时，数字的上下左右四个位置的大小约束也会限制四个方向后继位置的可行域。

在实现过程中，我们需要维护两个关键的数据结构。第一个是可行域 CurrentDomain 的概念，可行域是对于每一个节点而谈的，每当我们在某个位置放置一个数字时，同一行同一列以及周围位置的可行域都会随之发生变化。第二个是剪枝向量 prunedValues 的概念，剪枝向量也是对于每一个节点而谈的，当我们更改某个位置上的可行域时，万一该数字放置下去不能帮助我们找到最优解，就需要进行回溯操作，因此保存在每个位置被剪掉的数字也是必须的。

除此之外，由于可行域的存在，初始化函数将要求我们完成更多的事情，例如对于每个大小约束中若某个位置已经存在数字，我们就需要对该约束另一个位置的可行域进行一定的修正，行列也是同理，只要某个位置上存在数字，就需要对相同行和相同列的所有位置进行可行域修正。下面是运行结果的截图，和实验 PDF 中的结果是一致的。

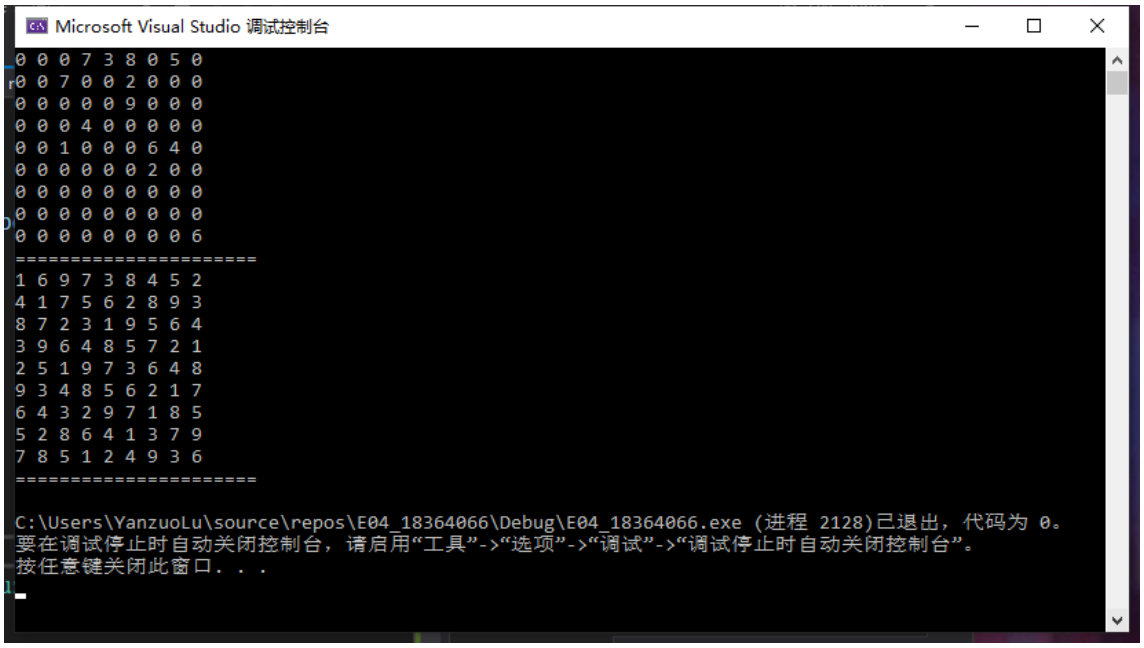

Figure 2: Forward Checking