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

E04 Futoshiki Puzzle (Forward Checking)

17341203 Yixin Zhang

September 22, 2019

Contents

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/.

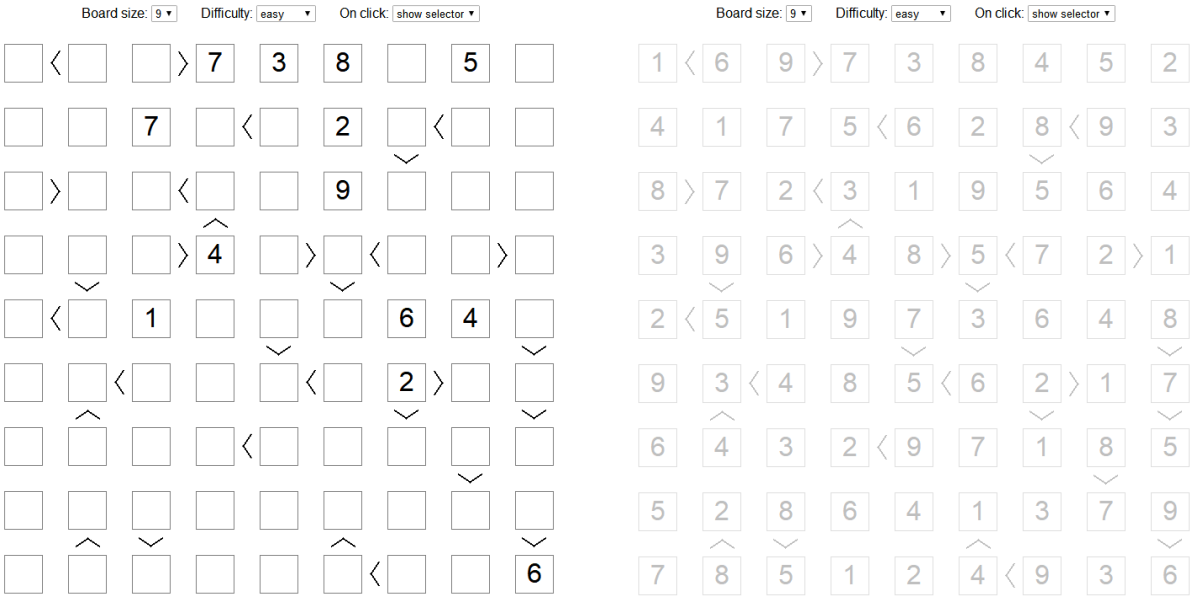

Figure 1: An Futoshiki Puzzle

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_201901@foxmail.com.

3 Basic Backtrack

3.1 Overview

Here I've tried to use basic backtrack algorithm to solve this problem. No optimization is made. I wrote this code because I want to make a comparison with forward checking algorithm.

3.2 Python Code

basic_backtrack.py

```python
# @Author: Jed Zhang
# @Date: 2019-09-19 16:33:50
# @Last Modified by: Jed Zhang
# @Last Modified time: 2019-09-19 16:33:50

import numpy as np
import copy

SIZE = -1


def isSolved(puzzle):
    """
    Check whether all cells in the puzzle is filled.
    """
    return puzzle.all() # test if all values are non-zero


def isValid(puzzle, constraints):
    """
    Check whether all constraints are satisfied.
    """
    s = set()

    # rows
    for i in range(SIZE):
        for j in range(SIZE):
            if puzzle[i, j] != 0 and puzzle[i, j] in s:
```

```
                return False # duplicated in row
            else:
                s.add(puzzle[i, j])
        s.clear()

    # columns
    for j in range(SIZE):
        for i in range(SIZE):
            if puzzle[i, j] != 0 and puzzle[i, j] in s:
                return False # duplicated in row
            else:
                s.add(puzzle[i, j])
        s.clear()

    # constraints
    for constraint in constraints:
        large = puzzle[constraint[0], constraint[1]]
        small = puzzle[constraint[2], constraint[3]]
        if large != 0 and small != 0 and not large > small:
            return False

    # pass all tests, so it is valid
    return True


def basicBacktrack(puzzle_origin, constraints):
    """
    Basic backtrack (without forward checking).
    If a solution is found, return it; else return None.
    """
    puzzle = puzzle_origin.copy() # the origin puzzle is not modified

    if isSolved(puzzle):
        return puzzle

    pos = np.where(puzzle == 0)
    pos = pos[0][0], pos[1][0] # position of the first empty cell
```

```
66
67      for i in range(1, SIZE + 1):
68          puzzle[pos] = i
69          if isValid(puzzle, constraints):
70              ret = basicBacktrack(puzzle, constraints)
71              if ret is not None:
72                  return ret
73
74      puzzle[pos] = 0 # restore the assignment
75      return None
76
77
78  def loadFutoshiki(puz_filename, con_filename):
79      """
80      Read the puzzle and constraints from files to numpy matrices,
81      and convert the coordinates into 0-indexed (coordinates in the
82      file are 1-indexed).
83      """
84      global SIZE
85      puzzle = np.loadtxt(puz_filename, dtype=np.uint8)
86      constraints = np.loadtxt(con_filename, dtype=np.uint8) - 1 # index start at 0 instead
            of 1
87      SIZE = len(puzzle) # the side length of the puzzle
88      return puzzle, constraints
89
90
91  if __name__ == '__main__':
92      # basic backtrack runs slowly, so we use a small puzzle (5x5) to test it
93      puzzle, constraints = loadFutoshiki('smallpuzzle.txt', 'smallconstraints.txt')
94      result = basicBacktrack(puzzle, constraints)
95
96      if result is not None:
97          print('Solution found:')
98          print(result)
99      else:
100         print('[-] No solution!')
```

3.3 Result

The code is able to output the correct solution on small puzzles. However, it takes forever to solve the problem at a bigger size, such as a 9*9 puzzle.

4 Forward Checking

4.1 Overview

Based on basic backtrack code above, I added forward checking and MRV into it. Every time a new variable is assigned, all values that violate the constraints will be removed from their domains.

Using the same idea, I implement the algorithm in both Python and C++. My purpose is still to compare.

Since this report is the second one (I have submitted one several hours ago), I would like to talk about the optimization that I've made in this newer version. I wrote a new function 'domainCount()', which counts the total number of values available in each variable's domain. While initializing the domains, I use a do-while loop to update domains several times, until the return value of 'domainCount()' cannot decrease anymore. With this procedure, the state space can be minimized before doing forward checking. According to the results, this optimization can make the calculation many times faster.

4.2 Python Code

python/forward_checking.py

```python
# @Author: Jed Zhang
# @Date: 2019-09-19 16:33:50
# @Last Modified by: Jed Zhang
# @Last Modified time: 2019-09-19 16:33:50

import numpy as np
import copy
from pprint import pprint

SIZE = -1


def isSolved(puzzle):
```

```
    """
    Check whether all cells in the puzzle is filled.
    """
    return puzzle.all() # test if all values are non-zero


def makeDomains(puzzle, constraints):
    """
    Make a dict as the initial domains of all variables. This function
    should be called only once at the beginning of the program.
    """

    def domainCount(domains):
        """
        Count the total number of values available in the puzzle.
        """
        count = 0
        for domain in domains.values():
            count += len(domain)
        return count

    # initialize
    domains = {}
    for i in range(SIZE):
        for j in range(SIZE):
            if puzzle[i, j] != 0:
                domains[i, j] = {puzzle[i, j]}
            else:
                domains[i, j] = set(range(1, SIZE + 1))

    # remove values that have conflict on rows or columns
    for i in range(SIZE):
        for j in range(SIZE):
            if puzzle[i, j] != 0:
                for i2 in range(SIZE):
                    if i2 != i and puzzle[i, j] in domains[i2, j]:
                        domains[i2, j].remove(puzzle[i, j])
```

```
                    if len(domains[i2, j]) == 0:
                        return None # DWO
            for j2 in range(SIZE):
                if j2 != j and puzzle[i, j] in domains[i, j2]:
                    domains[i, j2].remove(puzzle[i, j])
                    if len(domains[i, j2]) == 0:
                        return None # DWO

    # remove values that have conflict with constraints
    old_domain_count = 0
    while True:
        # repeat until total count of all domains cannot decrease anymore
        # I think this procedure can reduce the state space, thus speed up the search
        for constraint in constraints:
            large_pos = (constraint[0], constraint[1])
            small_pos = (constraint[2], constraint[3])
            if puzzle[large_pos] != 0: # large_pos has been assigned
                for i in range(puzzle[large_pos], SIZE + 1):
                    if i in domains[small_pos]:
                        domains[small_pos].remove(i)
                        if len(domains[small_pos]) == 0:
                            return None # DWO
            else: # large_pos has not been assigned
                minimum = min(domains[small_pos])
                if minimum in domains[large_pos]:
                    domains[large_pos].remove(minimum)


            if puzzle[small_pos] != 0: # small_pos has been assigned
                for i in range(1, puzzle[small_pos] + 1):
                    if i in domains[large_pos]:
                        domains[large_pos].remove(i)
                        if len(domains[large_pos]) == 0:
                            return None # DWO
            else: # small_pos has not been assigned
                maximum = max(domains[large_pos])
                if maximum in domains[small_pos]:
```

```
                    domains[small_pos].remove(maximum)

        # repeat ends
        new_domain_count = domainCount(domains)
        if new_domain_count == old_domain_count:
            break
        else:
            old_domain_count = new_domain_count


    return domains


def updateDomains(puzzle, constraints, domains_origin, pos):
    """
    In each iteration, we have chosen a pos using MRV, and assign a
    value in its domain to it. After that, we have to update some
    variables' domains by removing some values which has conflict with
    the assignment.
    """
    domains = copy.deepcopy(domains_origin) # deep copy

    # check the same column
    for i in range(SIZE):
        if i == pos[0]:
            continue
        if puzzle[i, pos[1]] == puzzle[pos]:
            return None
        if puzzle[pos] in domains[i, pos[1]]:
            domains[i, pos[1]].remove(puzzle[pos])
            if len(domains[i, pos[1]]) == 0:
                return None # DWO

    # check the same row
    for j in range(SIZE):
        if j == pos[1]:
            continue
```

```
125         if puzzle[pos[0], j] == puzzle[pos]:
126             return None
127         if puzzle[pos] in domains[pos[0], j]:
128             domains[pos[0], j].remove(puzzle[pos])
129             if len(domains[pos[0], j]) == 0:
130                 return None # DWO
131 
132     # check the constraints
133     for constraint in constraints:
134         large_pos = (constraint[0], constraint[1])
135         small_pos = (constraint[2], constraint[3])
136         if pos == large_pos:
137             for k in range(puzzle[pos], SIZE + 1):
138                 if k in domains[small_pos]:
139                     domains[small_pos].remove(k)
140                     if len(domains[small_pos]) == 0:
141                         return None # DWO
142         elif pos == small_pos:
143             for k in range(1, puzzle[pos] + 1):
144                 if k in domains[large_pos]:
145                     domains[large_pos].remove(k)
146                     if len(domains[large_pos]) == 0:
147                         return None # DWO
148     return domains
149 
150 
151 def mrv(puzzle, domains):
152     """
153     Find the variable with minimum remaining values (MRV),
154     and return its position.
155     """
156     min_val = SIZE * SIZE # max size of domain
157     min_pos = (-1, -1)
158     for i in range(SIZE):
159         for j in range(SIZE):
160             if puzzle[i, j] == 0 and len(domains[i, j]) < min_val:
161                 min_val = len(domains[i, j])
```

```
                min_pos = (i, j)
    return min_pos


def forwardChecking(puzzle_origin, constraints, domains_origin):
    """
    Use forward checking algorithm to solve the CSP problem.
    """
    puzzle = puzzle_origin.copy()
    domains = domains_origin.copy()

    if isSolved(puzzle):
        return puzzle

    pos = mrv(puzzle, domains) # find a unassigned variable using MRV

    for d in domains[pos]:
        puzzle[pos] = d
        temp_domains = updateDomains(puzzle, constraints, domains, pos)
        if temp_domains is not None: # not DWO
            ret = forwardChecking(puzzle, constraints, temp_domains)
            if ret is not None:
                return ret

    puzzle[pos] = 0 # restore the assignment
    return None


def loadFutoshiki(puz_filename, con_filename):
    """
    Read the puzzle and constraints from files to numpy matrices,
    and convert the coordinates into 0-indexed (coordinates in the
    file are 1-indexed).
    """
    global SIZE
    puzzle = np.loadtxt(puz_filename, dtype=np.uint8)
```

```
198     constraints = np.loadtxt(con_filename, dtype=np.uint8) - 1 # index start at 0 instead
            of 1
199     SIZE = len(puzzle) # the side length of the puzzle
200     return puzzle, constraints
201
202
203 if __name__ == '__main__':
204     puzzle, constraints = loadFutoshiki('../puzzle.txt', '../constraints.txt')
205     domains = makeDomains(puzzle, constraints)
206     result = forwardChecking(puzzle, constraints, domains)
207
208     if result is not None:
209         print('Solution found:')
210         print(result)
211     else:
212         print('[-] No solution!')
```

4.3 C++ Code

cpp/forward_checking.cpp

```
1  #include <fstream>
2  #include <iostream>
3  #include <map>
4  #include <set>
5  #include <string>
6  #include <vector>
7  using namespace std;
8
9  class Futoshiki {
10 public:
11     int size;
12     int con_num;
13     vector<vector<int>> puzzle;
14     vector<pair<pair<int, int>, pair<int, int>>> constraints;
15
```

```
16      Futoshiki(const char* puz_filename, const char* con_filename, int size, int con_num);
                // read puzzle and constraints from file
17      bool isSolved();
                                                                                    //
            check whether the puzzle is solved
18      vector<vector<set<int>>> makeDomains();
                                                                  // initialize the domains of
            each variable
19      vector<vector<set<int>>> updateDomains(vector<vector<set<int>>> domains, const
            pair<int, int>& pos); // update each domain after assigning a variable
20      pair<int, int> mrv(const vector<vector<set<int>>>& domains);
                                          // choose a unassigned variable with minimum
            remaining values
21      vector<vector<int>> forwardChecking(const vector<vector<set<int>>>& domains);
                        // try to solve the CSP
22
23  private:
24      // count the total number of values available in the puzzle
25      int domainCount(const vector<vector<set<int>>>& domains) {
26          int count = 0;
27          for(int i = 0; i < size; i++) {
28              for(int j = 0; j < size; j++) {
29                  count += domains[i][j].size();
30              }
31          }
32          return count;
33      }
34  };
35
36  /*
37      Read the puzzle and constraints from files to numpy matrices,
38      and convert the coordinates into 0-indexed (coordinates in the
39      file are 1-indexed).
40  */
41  Futoshiki::Futoshiki(const char* puz_filename, const char* con_filename, int size, int
        con_num)
42      : size(size), con_num(con_num), puzzle(size, vector<int>(size, 0)) {
```

```
43      ifstream puz_file(puz_filename), con_file(con_filename);
44      for (int i = 0; i < size; i++) {
45          for (int j = 0; j < size; j++) {
46              puz_file >> puzzle[i][j];
47          }
48      }
49
50      for (int i = 0; i < con_num; i++) {
51          int x1, y1, x2, y2;
52          con_file >> x1 >> y1 >> x2 >> y2;
53          constraints.push_back(make_pair(make_pair(x1 - 1, y1 - 1), make_pair(x2 - 1, y2 -
                1)));
54      }
55      puz_file.close();
56      con_file.close();
57  }
58
59  /* Check whether all cells in the puzzle is filled. */
60  bool Futoshiki::isSolved() {
61      for (int i = 0; i < puzzle.size(); i++) {
62          for (int j = 0; j < puzzle[0].size(); j++) {
63              if (puzzle[i][j] == 0) {
64                  return false;
65              }
66          }
67      }
68      return true;
69  }
70
71  vector<vector<set<int>>> Futoshiki::makeDomains() {
72      // initialize
73      vector<vector<set<int>>> domains(size, vector<set<int>>(size, set<int>()));
74      for (int i = 0; i < size; i++) {
75          for (int j = 0; j < size; j++) {
76              if (puzzle[i][j] == 0) {
77                  for (int k = 0; k < size; k++) {
78                      domains[i][j].insert(k + 1);
```

```
            }
        } else {
            domains[i][j].insert(puzzle[i][j]);
        }
    }
}

// remove values that have conflict on rows or columns
for (int i = 0; i < size; i++) {
    for (int j = 0; j < size; j++) {
        if (puzzle[i][j] != 0) {
            for (int i2 = 0; i2 < size; i2++) {
                if (i2 != i) {
                    domains[i2][j].erase(puzzle[i][j]);
                }
            }
            for (int j2 = 0; j2 < size; j2++) {
                if (j2 != j) {
                    domains[i][j2].erase(puzzle[i][j]);
                }
            }
        }
    }
}

// remove values that have conflict with constraints
int old_domain_count = 0, new_domain_count;
// repeat until total count of all domains cannot decrease anymore
do {
    for (int i = 0; i < con_num; i++) {
        pair<int, int> large_pos = constraints[i].first;
        pair<int, int> small_pos = constraints[i].second;
        if (puzzle[large_pos.first][large_pos.second] != 0) { // large_pos has been
            assigned
            for (int k = puzzle[large_pos.first][large_pos.second]; k <= size; k++) {
                domains[small_pos.first][small_pos.second].erase(k);
            }
```

```
115             }
116             else { // large_pos has not been assigned
117                 int minimum = *domains[small_pos.first][small_pos.second].begin();
118                 domains[large_pos.first][large_pos.second].erase(minimum);
119             }
120             if (puzzle[small_pos.first][small_pos.second] != 0) {
121                 for (int k = 1; k <= puzzle[small_pos.first][small_pos.second]; k++) {
122                     domains[large_pos.first][large_pos.second].erase(k);
123                 }
124             }
125             else {
126                 int minimum = *domains[large_pos.first][large_pos.second].rbegin();
127                 domains[small_pos.first][small_pos.second].erase(minimum);
128             }
129         }
130         new_domain_count = domainCount(domains);
131     } while(old_domain_count == new_domain_count);
132
133     return domains;
134 }
135
136 /*
137     In each iteration, we have chosen a pos using MRV, and assign a
138     value in its domain to it. After that, we have to update some
139     variables' domains by removing some values which has conflict with
140     the assignment.
141 */
142 vector<vector<set<int>>> Futoshiki::updateDomains(vector<vector<set<int>>> domains, const
    pair<int, int>& pos) {
143     // check the same column
144     for (int i = 0; i < size; i++) {
145         if (i == pos.first)
146             continue;
147         else if (puzzle[i][pos.second] == puzzle[pos.first][pos.second]) {
148             return vector<vector<set<int>>>(); // DWO
149         } else {
150             domains[i][pos.second].erase(puzzle[pos.first][pos.second]);
```

```
            if (domains[i][pos.second].size() == 0) {
                return vector<vector<set<int>>>(); // DWO
            }
        }
    }

    // check the same row
    for (int j = 0; j < size; j++) {
        if (j == pos.second)
            continue;
        else if (puzzle[pos.first][j] == puzzle[pos.first][pos.second]) {
            return vector<vector<set<int>>>(); // DWO
        } else {
            domains[pos.first][j].erase(puzzle[pos.first][pos.second]);
            if (domains[pos.first][j].size() == 0) {
                return vector<vector<set<int>>>(); // DWO
            }
        }
    }

    // check the constraints
    for (int i = 0; i < con_num; i++) {
        pair<int, int> large_pos = constraints[i].first;
        pair<int, int> small_pos = constraints[i].second;
        if (pos == large_pos) {
            for (int k = puzzle[pos.first][pos.second]; k <= size; k++) {
                domains[small_pos.first][small_pos.second].erase(k);
                if (puzzle[small_pos.first][small_pos.second] == 0 &&
                    domains[small_pos.first][small_pos.second].size() == 0) {
                    return vector<vector<set<int>>>(); // DWO
                }
            }
        } else if (pos == small_pos) {
            for (int k = 1; k <= puzzle[pos.first][pos.second]; k++) {
                domains[large_pos.first][large_pos.second].erase(k);
                if (puzzle[large_pos.first][large_pos.second] == 0 &&
                    domains[large_pos.first][large_pos.second].size() == 0) {
```

```
                    return vector<vector<set<int>>>(); // DWO
                }
            }
        }
    }
    return domains;
}

/*
    Find the variable with minimum remaining values (MRV),
    and return its position.
*/
pair<int, int> Futoshiki::mrv(const vector<vector<set<int>>>& domains) {
    int min_val = size * size; // max size of domain
    pair<int, int> min_pos = make_pair(-1, -1);
    for (int i = 0; i < size; i++) {
        for (int j = 0; j < size; j++) {
            if (puzzle[i][j] == 0 && domains[i][j].size() < min_val) {
                min_val = domains[i][j].size();
                min_pos = make_pair(i, j);
            }
        }
    }
    return min_pos;
}

/* Use forward checking algorithm to solve the CSP problem. */
vector<vector<int>> Futoshiki::forwardChecking(const vector<vector<set<int>>>& domains) {
    if (isSolved()) {
        return puzzle;
    }

    pair<int, int> pos = mrv(domains);

    for (auto pd = domains[pos.first][pos.second].begin(); pd !=
        domains[pos.first][pos.second].end(); pd++) {
        puzzle[pos.first][pos.second] = *pd;
```

```
222         auto temp_domains = updateDomains(domains, pos);
223         if (temp_domains.size() != 0) { // not DWO
224             vector<vector<int>> ret = forwardChecking(temp_domains);
225             if (ret.size() != 0) return ret;
226         }
227     }
228 
229     puzzle[pos.first][pos.second] = 0; // restore the assignment
230     return vector<vector<int>>();
231 }
232 
233 int main() {
234     Futoshiki game("../puzzle.txt", "../constraints.txt", 9, 30);
235     auto domains = game.makeDomains();
236     vector<vector<int>> result = game.forwardChecking(domains);
237     if (result.size() != 0) {
238         cout << "Solution found:" << endl;
239         for (int i = 0; i < game.size; i++) {
240             for (int j = 0; j < game.size; j++) {
241                 cout << result[i][j] << " ";
242             }
243             cout << endl;
244         }
245     } else {
246         cout << "[-] No solution!" << endl;
247     }
248     return 0;
249 }
```

4.4 Result

Both Python implementation and C++ implementation can produce correct result, but C++ is much faster than Python. The Python code takes about 25 seconds to solve a 9*9 puzzle, while the C++ code takes only 1 seconds.

```
$ time python3 forward_checking.py
Solution found:
[[1 6 9 7 3 8 4 5 2]
 [4 1 7 5 6 2 8 9 3]
 [8 7 2 3 1 9 5 6 4]
 [3 9 6 4 8 5 7 2 1]
 [2 5 1 9 7 3 6 4 8]
 [9 3 4 8 5 6 2 1 7]
 [6 4 3 2 9 7 1 8 5]
 [5 2 8 6 4 1 3 7 9]
 [7 8 5 1 2 4 9 3 6]]
python3 forward_checking.py  24.47s user 0.36s system 98% cpu 25.222 total
```

```
$ g++ forward_checking.cpp && time ./a.out
Solution found:
1 6 9 7 3 8 4 5 2
4 1 7 5 6 2 8 9 3
8 7 2 3 1 9 5 6 4
3 9 6 4 8 5 7 2 1
2 5 1 9 7 3 6 4 8
9 3 4 8 5 6 2 1 7
6 4 3 2 9 7 1 8 5
5 2 8 6 4 1 3 7 9
7 8 5 1 2 4 9 3 6
./a.out  1.34s user 0.02s system 97% cpu 1.397 total
```

5 Discussion

After four experiments, I can clearly feel the difference in speed between Python and C++. Writing Python code is definitely a pleasure, but it is really slow. C++ is faster, but you need to pay more attention while writing code, especially when using STL.

The optimazation that I have made in this second version is really effective. It makes the algorithm about 50 times faster. Actually, the searching procedure has not been optimized. It is the initial state space that has been minimized.