CM1035: Algorithms & Data Structures I

Midterm Assignment

Arjun Muralidharan

30th December 2019

Contents

Note on Notation In this assignment, I use consistent notation for functions and arguments as FunctionName(*arugment1, argument2, ...*). For data structure, notation of $v[k]$ is used. Operations on data structures are treated as functions, such as Dequeue(q).

Assumptions We are designing an algorithm strictly for a 4×4 pseudoku as outlined in the assignment. The algorithms therefore assume fixed row and columns lengths of 4, and a fixed number of 16 squares used in the puzzle.

1 MakeVector

Algorithm 1

function MakeVector(*row*)
  new Vector $puzzle(4)$
  for $1 \le i \le 4$ **do**
    $puzzle[i] \leftarrow row$
  end for
  return *puzzle*
end function

2 PermuteVector

Algorithm 2

function PermuteVector(*row, p*)
  new Queue q
  for $i \leftarrow 4$ **downto** 1 **do**
    Enequeue($row[i], q$)
  end for
  for $1 \le i \le p$ **do**
    $store \leftarrow$ HEAD(q)
    Dequeue(q)
    Enqueue($store, q$)
  end for
  return q
end function

3 PermuteRow

Algorithm 3

```
function PermuteRow(puzzle, x, y, z)
    puzzle[1] ← PermuteVector(puzzle[1], x)
    puzzle[2] ← PermuteVector(puzzle[2], y)
    puzzle[3] ← PermuteVector(puzzle[3], z)
    return puzzle
end function
```

4 CheckColumn

Algorithm 4

```
function CheckColumn(puzzle, j)
    new Vector temp(4)
    for 1 ≤ i ≤ 4 do
        temp[i] ← puzzle[i][j]
    end for
    for 1 ≤ k ≤ 4 do
        if LinearSearch(temp, k) = false then
            return false
        end if
    end for
    return true
end function
```

5 ColCheck

Algorithm 5

```
function ColCheck(puzzle)
    for 1 ≤ i ≤ 4 do
        if CheckColumn(puzzle, j) = false) then
            return false
        end if
    end for
    return true
end function
```

6 CheckGrids

We assume that the MakeGrid and LinearSearch functions are declared and work as defined in the assignment.

Algorithm 6

```
function CheckGrids(puzzle)
    new Vector grids(4)
    grids[1] ← MakeGrid(puzzle, 1, 1, 2, 2)
    grids[2] ← MakeGrid(puzzle, 1, 3, 2, 4)
    grids[3] ← MakeGrid(puzzle, 3, 1, 4, 2)
    grids[4] ← MakeGrid(puzzle, 3, 3, 4, 4)
    for 1 ≤ i ≤ 4 do
        for 1 ≤ k ≤ 4 do
            if LinearSearch(grids[i], k) = false then
                return false
            end if
        end for
    end for
    return true
end function
```

7 MakeSolution

My implementation aims to provide a selection out of all possible solutions at random. A simpler implementation would just return the first solution found, however this would always result in the same solution since the mechanism to cycle through all permutations is deterministic.

Instead, for generating a valid solution, we loop through all possible cyclical permutations of rows until we find such solutions. We store all solutions as nested vectors in a dynamic array **solutions**, of which one is returned at random as the assignment requires returning only one solution. We additionally define a function RandomInt*(min, max)* which generates a pseudorandom integer between x and y. This is used to pick one of the valid solutions and return this.

Algorithm 7

```
function MakeSolution(row)
    puzzle ← MakeVector(row)
    new Dynamic Array solutions(0)
    for 0 ≤ x ≤ 3 do
        for 0 ≤ y ≤ 3 do
            for 0 ≤ z ≤ 3 do
                candidate ← PermuteRow(puzzle, x, y, z)
                if ColCheck(candidate) = true ∧ CheckGrids(candidate) = true then
                    solutions[Length(solutions) + 1] ← candidate
                end if
            end for
        end for
    end for
    selectedSolution ← RandomInt(1, Length(solutions))
    return solution[selectedSolution]
end function
```

8 Puzzle Creation Method

To create a puzzle with blanks, I would proceed with the following steps.

1. Check if n is a valid input, i.e. it is at least 1 and at most the number of squares in the grid
2. For an even distribution of blanks, we first loop through each row for n iterations. If n is greater than the number of rows, we repeat the process over all rows again until we have iterated n times
3. For each loop iteration, we loop over the individual elements of the row and remove one random element from the row, using a pseudorandom number generator function.
4. We return the puzzle with removed elements.

We again assume that we have a pseudorandom integer generator at hand, namely RandomInt*(min, max)*.

The function CreatePuzzle(*puzzle, n*) takes a puzzle of the format of the output of MakeSolution and returns a puzzle of the same format with blanks inserted in n locations.

We further use modulo operations to cycle over the individual rows of the puzzle.

Algorithm 8

```
function CreatePuzzle(puzzle, n )
    if n < 1 ∨ n > 16 then
        return false
    end if
    for 1 ≤ i ≤ n do
        randSlot ←RandomInt()1, 4)
        while puzzle[i mod 4][randSlot] = ∅ do
            randSlot = (randSlot mod 4) + 1
        end while
        puzzle[i mod 4 + 1][j] ← ∅
    end for
    return puzzle
end function
```

9 Puzzles not possible with the current algorithm

As the current algorithm relies on cyclical permutations of an input vector, any puzzle where the row vectors are not cyclical permutations of each other could not be generated using this puzzle.

An example of such a puzzle would be shown in Table 1.

3	1	4	2
2	4	1	3
4	2		1
1	3	2	4

Table 1. *Example of an alternative pseudoku*

10 Alternative approach to generating puzzles

A simple alternative to the algorithm outlined so far would be to not use cyclical permutation, but some other way to manipulate the input vector to generate rows.

One method would be:

1. Store the input vector in a dynamic array
2. Use a shuffle method (e.g. **Fisher-Yates-Durstenfeld**) to find a pseudorandom permutation of a row vector
3. Create new puzzles indefinitely until a puzzle satisfying the conditions is found

This process would affect primarily the PermuteVector and MakeSolution functions. MakeSolution function needs to be adapated to loop indefinitely until a solution is found. This may be more computationally intensive, but yields many more possible solutions than the previous algorithm.

The function PermuteRow is almost identical with the omission of the parameters x, y and z.

We use the **Durstenfeld Shuffle** to randomly permute the input vectors until a solution is found.[1]

Algorithm 9 Alternative for PermuteVector

function PermuteVector(*row*)
  new Dynamic Array d
  for $1 \leq i \leq 4$ **do**
    $d[\text{Length}(d) + 1] \leftarrow row[i]$
  end for
  for $i = \text{Length}(d) - 1$ **downto** 2 **do**
    $j \leftarrow \text{RandomInt}(1, i)$
    $temp \leftarrow d[i]$
    $d[i] \leftarrow d[j]$
    $d[j] \leftarrow temp$
  end for
  return d
end function

Algorithm 10 Alternative for MakeSolution

function MakeSolution(*row*) $puzzle \leftarrow \text{MakeVector}(row)$
  while $\text{ColCheck}(candidate) = \texttt{false} \lor \text{CheckGrids}(candidate) = \texttt{false}$ **do**
    $puzzle \leftarrow \text{PermuteRow}(puzzle)$
  end while
  return puzzle
end function

[1] Richard Durstenfeld. 1964. Algorithm 235: Random permutation. Commun. ACM 7, 7 (July 1964), 420. DOI:https://doi.org/10.1145/364520.364540