IIC2133 - T02

Jonathan Lee

11 October 2018

1 Analysis

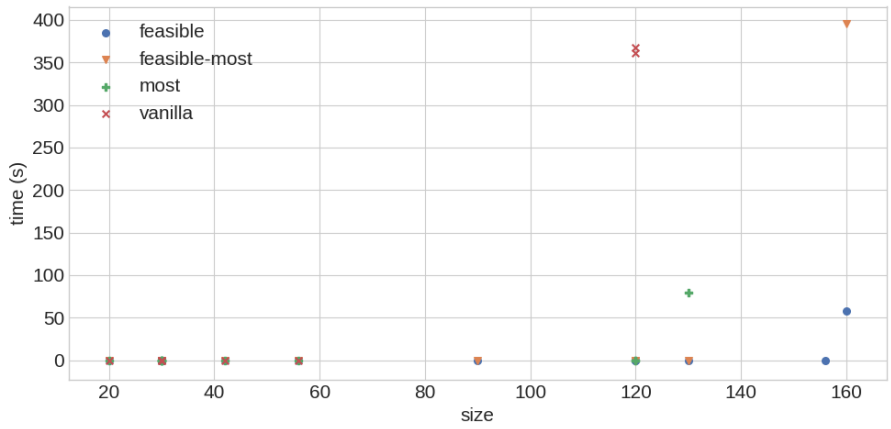

Figure 1: Scatter plot puzzle size as $N \times M$ and computation time in seconds

Figure 1 compares the puzzle size with the computation time under four variants of the program. All the experiments have been performed with a 10 minutes time limit, i.e., the program is stopped at the 10 minute mark. The *vanilla* variant without any optimization maintains a computation time of less than 1 second until size 120 before increasing to 350 seconds (6 minutes) and then failing to solve any larger puzzles. This suggests an exponential complexity.

Similarly, the other three variants show similar performance behavior where computation time is very low until a certain point where it incurs a huge increase in computation time. This suggests that ultimately the algorithm has an exponential complexity with respect to data size.

However, optimization techniques clearly push back the point at which computation time shoots up. Most notably, the *feasible* variant which applies the *feasible* prune strategy is able to solve all puzzles in almost insignificant times, except for the last one which took 50 seconds. Details of the prune strategies

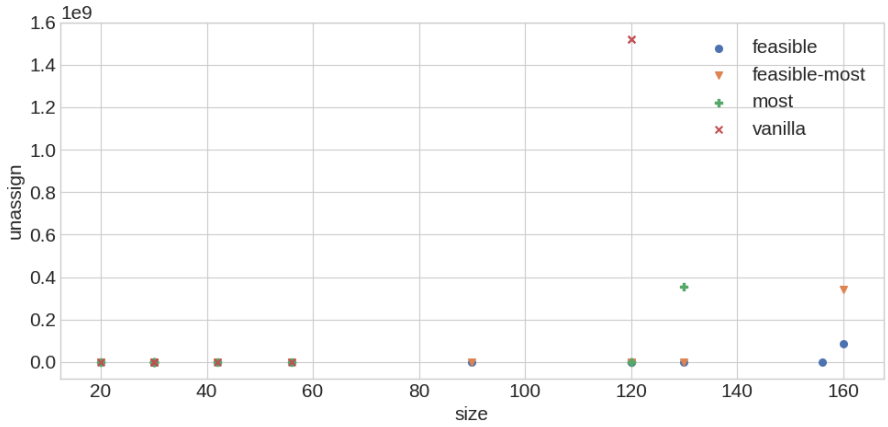

Figure 2: Scatter plot puzzle size as $N \times M$ and number of unassignments

are explained in Section 2.2. In contrary, combining with the heuristic that assigns cells with the most constraints first does not improve performance significantly. In fact, it can actually harm performance when combined with the *feasible* pruning strategy as shown by the puzzle of size 160 where the variant *feasible-most* took 400 seconds (6.5 minutes) as compared to the *feasible* variant which took 50 seconds.

Figure 2 compares the puzzle size with the number of unassignments under four variants of the program. Similar to Figure 1, the number of unassignments skyrockets at a certain point. This again suggests an exponential complexity. For the *variant*, the number of unassignments goes up to 1.6 billion at puzzle size 120. In contrary, even with the largest puzzle of size 160, the *feasible* variant still has less than 10% of the assignments.

2 Optimization

In the following possible optimization techniques are presented; two optimization categories have been implemented to achieve significant performance improvement.

2.1 Note on implementation

While the straightforward way is to go through all the cells in the board, it is redundant to go through both cell type pairs *top*/*bottom* and *left*/*right* since an assignment of charge $x \in \{-1, 0, 1\}$ to a *top* cell immediately determines its *bottom* counterpart to have charge $y = -x$. Therefore, to manage the cells in my implementation, there is

1. A matrix of board cells as an array of Cell pointer arrays

2. Row and column counters as 2 integer array
3. Cell variables of type *top* and *left* to loop through during backtracking as cell pointer array of size $r \times c/2$
4. Counter to keep track of assigned variables as a boolean array of size $r \times c/2$
5. Counter integer of assigned variables to make termination condition check $\mathcal{O}(1)$

The above means that when at the "safety" check and assignment of cell $a_{i,j}$, its counterpart $a_{i+1,j}$ or $a_{i,j+1}$ is also checked and assigned. For simplicity, please assume this detail in the following unless specified otherwise.

For the constraints, 8 integer arrays are used to store the required constraints and the current number of charges from the assigned cells.

2.2 Pruning

Idea

At insertion of a cell $a_{i,j}$ with charge x, pruning is done to check whether if this assignment will permit the fulfillment of the $+\&-$ row and column constraints in the future. By this, we avoid exploring dead ends that do not ultimately fulfill the required constraints before trying to fill in all the cells in a row/col for restriction evaluation.

For example, if following cell assignment of negative charge, the number of remaining possible positive cell candidates on row i is $\text{cand}_i^+ = 4$ and the positive row constraint $\text{constr}_i^+ = 5$, then this cell assignment should be pruned since the constraint will not be fulfilled in the future.

But, how to compute the number of cell candidates? A straightforward version is the number of remaining unassigned cells. This has been implemented as the **sufficient** prune strategy. However, we note that for an interval of l cells, the maximum number of positive or negative charges is not l since two adjacent cells cannot have the same non-empty charge. Rather, $\text{cand}_i^+ = \lceil l/2 \rceil$. Furthermore, each of the candidate requires valid neighbors, that is an unassigned cell cannot be a positive charge candidate if it has a positive cell neighbor. In the **feasible** prune strategy, these two concepts are incorporated and achieves significant performance gains.

Implementation

Sufficient prune strategy is a straightforward comparison between the number of remaining required charges (constr $-$ charge) and remaining unassigned cells (size $-$ assigned) and then prune the assignment if the number of remaining required charges is negative or larger than the number of remaining unassigned cells. There are also several details such as checking the constraint is non-zero. This is essentially 4 checks and therefore $\mathcal{O}(1)$.

In contrary, **feasible** is not $\mathcal{O}(1)$ but $\mathcal{O}(n)$ where $n = max(row, col)$. This is because at checking, intervals of valid unassigned cells have to be identified. The number of intervals is $\mathcal{O}(n)$ since in the worst case half of the row/col is assigned, leaving interleaving intervals of size one or two. Interval validity check is implemented by looping over a row/col and stopping at assigned cells or invalid empty cells to compute $\lceil l/2 \rceil$ where l is the interval size. Each cell has a boolean to indicate if it has been assigned a value. In addition, a cell has the assigned value so that neighbor value check can be done in constant time by accessing neighbor cells from the cell pointer matrix. Once the total number of available charge spaces is computed, the cell assignment is pruned if the remaining required charges is negative or larger than the total number of available charge spaces.

Experiments

As shown in the previous section, we achieved the best results using just the **Feasible** prune strategy. Only puzzle *test_09.txt* takes 1 minute to complete, all other puzzles take less than 1 second to solve.

2.3 Heuristics

Idea

Heuristics can be applied in the ordering of the variables to explore during backtracking and the ordering of values to explore of a particular variable. The idea is that it is desirable to explore variables with the most constraints first because their domain spaces are much smaller and therefore have a higher probability of getting the correct value. In contrary, for the ordering of values per variable, it is typically desirable to explore less restrictive value first so that we do not significantly shrink the domain of unassigned variables. For this puzzle problem, a **most_constraint** heuristics orders cells by the sum of its constraints in a decreasing order. This means that we explore cells with more constraints first. For each variable, since the *empty* charge is the least restrictive, we always assign charge value in the order *empty*, *positive*, and *negative*. This simpler approach seemed to yield better performance than more complicated approaches following evaluation.

Implementation

As mentioned in Section 2.1, the backtracking function loop through the extra list of cell pointers rather than the matrix of cell pointers. This means that prior to calling the backtracking function, the list of cell pointers is sorted by merge sort using a priority of the cell in $\mathcal{O}(nlgn)$ one time. This is trivial since mounting the board takes $\mathcal{O}(n^2)$. The priority value of a cell is computed as the sum of both the cell's and its counterpart's constraints. During backtracking, an integer index is used to denote the cell whose value we are trying

to assign. To implement individual value ordering per cell, we simply have each cell containing a list of possible values where each value again has a priority value. Similar to the cell ordering, this list can be sorted and an integer index is used to identify the value that is being evaluated at a certain point in time.

Experiments

Since the value domain has only three values, we found that there is not much gain in putting the values in a particular order. In contrary, the *most_constraint* has mixed effects on performance. If no pruning is used, using this heuristics can significantly improve performance. For example, it takes 6 minutes to solve *test_06* without any optimization and it takes 0.06 seconds with the application of the **most_constraint** heuristics. However, when combined with the **feasible** pruning strategy, *test_08* cannot be solved within 10 minutes, whereas with only the **feasible** pruning strategy, the same puzzle takes 0.09 seconds.

2.4 Propagation

Idea

Following a cell assignment $a_{i,j}$ with charge x, forward propagation assigns several more cells before calling the recursive backtracking function again. Then if the recursive call returns false, charge x is unassigned from $a_{i,j}$ and all assignments done in forward propagation are undone as well. This fits with the **feasible** pruning strategy so that if the number of available charge spaces is greater or equal to the number of remaining required charges of a particular constraint, the cells corresponding to the available charge space can be assigned in one go to meet the constraint. In this way, the exploration is done in a much faster manner.

Implementation

Since there is an explicit array of cell pointers that gets iterated through during the recursive backtracking function, forward propagation involves assigning several of the cells and then changing their status from not assigned to assigned in both the cell's assigned boolean and the array of assignment booleans that mirrors the explicit array of cell pointers. Then, a slight modification has to be made to the Cell * get_next_cell(Puzzle *p, int k) function. Instead of just returning the kth cell, it has to return the next unassigned cell starting from index k because $k+1, \ldots, n$ might be assigned from a previous forward propagation call. This means that Cell * get_next_cell(Puzzle *p, int k) is no longer a constant time operation. To take care of the unpropagation, in the recursive bool r_backtrack(Puzzle *p, int k), an extra integer array has to be allocated so that if the next call r_backtrack(p, k + 1) returns false, we can unassign all the propagated cells whose positions are stored in the extra integer array. With regards to the actual propagation of each cell, this is basically the same as the

typical cell charge assignment. We can work out the specific charge to assign by using the already assigned $a_{i,j}$. If $a_{i,j}$ has charge $x \in \{-1, 1\}$, then we should assign the adjacent cell with charge $-x$ unless the remaining charge constraint is 0.