**Marine Control Systems**

**Main engine auxiliary system - water cooling.**

1. **Intoduction**

The purpose of the task was to create the program which solves Sudoku problem. There are four examples to solve (figure 1).

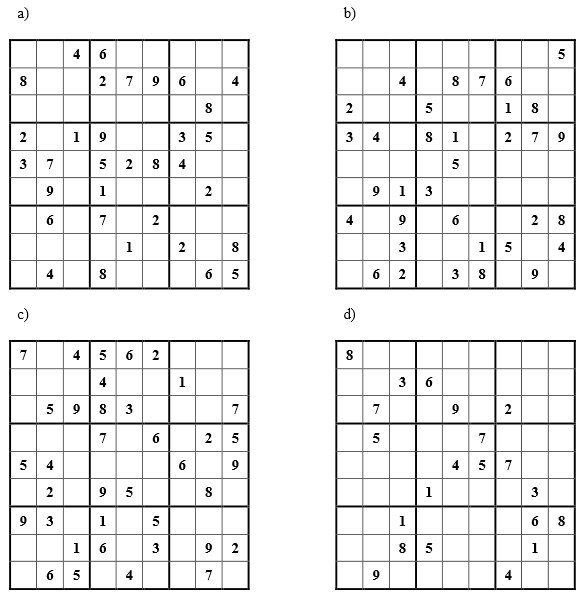
**

Figure 1. Sudoku examples to solve.

1. **Sudoku**

Sudoku is a logic-based, combinatorial number-placement puzzle. The objective is to fill a 9×9 grid with digits so that each column, each row, and each of the nine 3×3 sub-grids that compose the grid (also called "boxes", "blocks", "regions", or "sub-squares") contains all of the digits from 1 to 9. The puzzle setter provides a partially completed grid, which for a well-posed puzzle has a unique solution.

Completed puzzles are always a type of Latin square with an additional constraint on the contents of individual regions. For example, the same single integer may not appear twice in the same row, column or in any of the nine 3×3 subregions of the 9x9 playing board.

An example of solved Sudoku is shown in the figure 2.



Figure 2. An example of solved Sudoku

There are different algorithms used to solving Sudoku problem, for example branch and bound algorithm and brute-force search method.

1. **Branch and Bound**

Branch and Bound is a general method for finding optimal solutions to problems, typically discrete problems. A branch and bound algorithm searches the entire space of candidate solutions. It throws out large parts of the search space by using previous estimates on the quantity being optimized. The method is due to Ailsa Land and Alison Doig, who wrote the original paper in 1960: “An automatic method of solving discrete programming problems.”

Typically we can view a branch-and-bound algorithm as a tree search. At any node of the tree, the algorithm must make a finite decision and set one of the unbound variables. Suppose that each variable is only 0-1 valued. Then at a node where the variable {x} is on-tap, the algorithm must decide whether or not {x=0} or {x=1} is the right choice. In a brute-force search both choices would be examined.

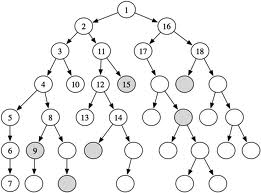
[](https://rjlipton.wordpress.com/2012/12/19/branch-and-bound-why-does-it-work/tree-6/)

Figure 3.Branch and Bound method.

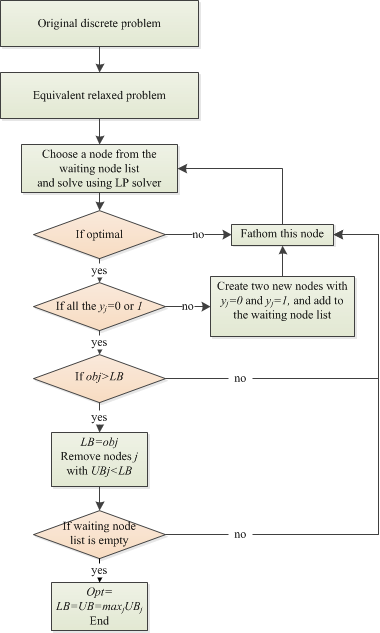


Figure 4. The flow chart for Branch and Bound algorithm

In the Branch and Bound case the algorithm tries to avoid searches that are useless and this is the power of the method.

1. **Brute-force search**

Brute-force is a very general problem-solving technique that consists of systematically enumerating all possible candidates for the solution and checking whether each candidate satisfies the problem's statement.

A brute-force algorithm to find the divisors of a natural number *n* would enumerate all integers from 1 to the square root of n, and check whether each of them divides *n* without remainder. A brute-force approach for the eight queens puzzle (the problem of placing eight chess queens on an 8×8 chessboard so that no two queens threaten each othe) would examine all possible arrangements of 8 pieces on the 64-square chessboard, and, for each arrangement, check whether each (queen) piece can attack any other.

A brute-force is simple to implement and will always find a solution if it exists but its cost is proportional to the number of candidate solutions. Therefore, brute-force search is typically used when the problem size is limited, or when there are problem-specific heuristics that can be used to reduce the set of candidate solutions to a manageable size. The method is also used when the simplicity of implementation is more important than speed.

In order to apply brute-force search to a specific class of problems, one must implement four procedures, *first*, *next*, *valid*, and *output*. These procedures should take as a parameter the data *P* for the particular instance of the problem that is to be solved, and should do the following:

1. *first* (*P*): generate a first candidate solution for *P*.
2. *next* (*P*, *c*): generate the next candidate for *P* after the current one *c*.
3. *valid* (*P*, *c*): check whether candidate *c* is a solution for *P*.
4. *output* (*P*, *c*): use the solution *c* of *P* as appropriate to the application.