ჩემი სუდოკუს ამომხსნელის ფსევდოკოდი:

```
main():
    read sudokus from file specified in arguments
    pass one sudoku at a time to solve()
    write returned solution to file specified in arguments

solve(sudoku):
    initialize domains as a list of lists of 1,2,...,9
    update domains according to the filled part of given sudoku
    recursiveSolve(sudoku as a list, domains, mrv(sudoku, domains))
    return solution as a string, if there is one

recursiveSolve(sudoku, domains, index):
    if no .-s left in sudoku, it's solved and return true
    for every value from domain of cell with given index:
        write this value in this cell
        update domains according to this new value
        if update leaves any cell with an empty domain:
            return to old domains
            write . in given cell again
            move to the next value in domain
        if recursiveSolve(sudoku, domains, mrv(sudoku, domains)) returned true:
            it's solved and return true
        return to old domains
        write . in given cell again
    return false, in case no value worked

mrv(sudoku, domains):
    for every cell:
        if it's not yet filled and has domain smaller than we've yet seen:
            update info about cell with minimum remaining value
    return index of cell with minimum remaining value

updateDomains(domains, val, index, changedDomainIndexes):
    add this cell's current domain to changedDomainIndexes
    set this cell's domain to only val
    get row and column of cell using index
    for every cell in this row/column/square but cell with given index:
        if its domain contains val:
            if it only contains val:
                return false
            remove val from domain
            add its index to changedDomainIndexes

returnToOldDomains(domains, changedDomainIndexes, val, index):
    set domain of cell with given index to first element of changedDomainIndexes
    for every cell in changedDomainIndexes:
        add val to this cell's domain
```

ჩემს იმპლემენტაციაში სუდოკუ წარმოდგენილია, როგორც ერთ განზომილებიანი list, რომელიც გადანომრილია ზედა მარცხენა უჯრიდან ქვედა მარჯვენა უჯრამდე. თითოეული უჯრა არის ცვლადი და აქვს თავისი დომეინი. დომეინები არის list-ების list, სადაც domains[i] წარმოადგენს i-ური ინდექსის მქონე უჯრის დომეინს. Set-ების list-ც ვცადე, რაც უფრო ლოგიკური მეჩვენებოდა, თუმცა, დრო არ გააუმჯობესა,

პირიქით, საშუალოდ 17 წამში რომ ხსნიდა 1011-ვე სუდოკუს, ამ ცვლილების შემდეგ საშუალოდ 19 წამზე ავიდა და დავაბრუნე ძველი. შეზღუდვების დაკმაყოფილების უზრუნველსაყოფად ვიყენებ forward checking-ს - ყოველი მნიშვნელობის ჩასმამდე ვამოწმებ ეს მნიშვნელობა რომელიმე მასთან ორობითი შეზღუდვით დაკავშირებული უჯრის დომეინს ცარიელს ჰო არ ტოვებს, და თუ ტოვებს, არ ვსვამ ამ მნიშვნელობას. ორობითი შეზღუდვებით დაკავშირებული თითოეულ უჯრასთან არიან მის სტრიქონში, სვეტში და კვადრატში მყოფი უჯრები. რადგან სუდოკუ ერთ განზომილებიან list-ში მაქვს შენახული, სტრიქონზე, სვეტზე და კვადრატზე გადასაყოლად ჯერ სტრიქონისა და სვეტის ინდექსებს ვიგებ უჯრის ინდექსიდან. თუ რომელიმე უჯრაზე მისი დომეინიდან ყველა მნიშვნელობა ვცადე და არცერთმა არ მიმიყვანა სწორ ამოხსნამდე, backtracking-ით ვბრუნდები მის წინა უჯრაზე და სხვა მნიშვნელობას ვსვამ.

Forward checking-ის დროს დომეინებს ვააფდეითებ, იგივე სტრიქონში, სვეტში და კვადრატში მყოფი უჯრების დომეინებიდან ვშლი იმ მნიშვნელობას, რომელიც ჩავსვი ახლა კონკრეტულ უჯრაში, ხოლო backtracking-ის დროს აუცილებელია დომეინების ისევ ძველ მნიშვნელობებზე დაბრუნება, ამიტომ updateDomains მეთოდს გადაეცემა list (changedDomainIndexes), რომელშიც იწერება ყველა იმ უჯრის ინდექსი, რომლის დომეინიდანაც წაიშალა ეს მნიშვნელობა, რადგან შეიძლება ზოგი უჯრის დომეინში საერთოდ არ ყოფილა ეს მნიშვნელობა, ხოლო backtracking-ის დროს გადავყვები ამ ინდექსებს და ყველგან ისევ ვამატებ მას. დომეინების დააფდეითებისას ასევე რომელ უჯრასაც ვავსებ, იმის დომეინს ვხდი მხოლოს ახალ ჩაწერილ მნიშვნელობას, ამიტომ changedDomainIndexes-ში ვამატებ ამ უჯრის ძველ დომეინს, რათა backtracking-სას აღდგენა შევძლო. თავიდან ამ ყველაფრის მაგივრად უბრალოდ ყოველ ჯერზე deepcopy-თი ვაკოპირებდი დომეინებს და ისე გადავცემდი რეკურსიას, თუმცა, ეს ძალიან ცუდი იყო დროის მხრივ, საშუალოდ 370 წამს ანდომებდა ყველა სუდოკუს ამოხსნას, როცა ამის შეცვლის შემდეგ 17 წამზე ჩამოვიდა.

ამ csp-ის ამოსახსნელად პირველი რაც მოვიფიქრე და დავწერე იყო მხოლოდ forward checking, რაც ფაილში მოცემულ პირველ 2 სუდოკუს შედარებით ადეკვატურ დროში ხსნიდა, თუმცა, მე-3-ს ამოხსნას საათზე მეტს ანდომებდა, ამიტომ ცხადი გახდა, რომ არ იყო საკმარისი. Arc consistency თავიდანვე არალოგიკურად მეჩვენებოდა სუდოკუსთვის, რადგან ყოველი უჯრის შევსების შემდეგ, ყველა მისი  გრაფში მეზობლის დომეინის განხილვა გახდებოდა საჭირო ამ ახალ მნიშვნელობასთან მიმართებაში, ხოლო ამ შემოწმებებისას რომელიმეს დომეინი თუ შეცვლებოდა, მისი მეზობლების (csp-ს გრაფში მეზობლებს ვგულისხმობ, ანუ შეზღუდვით დაკავშირებულს მასთან) ამ უჯრასთან მიმართებაში განხილვაც დაგვჭირდებოდა, რაც ძალიან ბევრ რესუსრსს წაიღებდა, ხოლო შედეგი forward checking-ზე ბევრად

უკეთესი არ ექნებოდა, რადგან უჯრის შევსება ძირითადად მხოლოდ მისი მეზობლების (csp-ს გრაფში) დომეინებზე ახდენს გავლენას. თუმცა, ამის მიუხედავად, მაინც ვცადე arc consistency, როგორც ცალკე, ისე მხოლოდ mrv-სთან (minimum remaining value), მხოლოდ lcv-სთან (least constraining value) და ორივესთან ერთად. თუმცა შედეგები ცალსახად ჩამოუვარდებოდა forward checking-ის შედეგებს იმავე პირობებში, ამიტომ forward checking დავაბრუნე.

Forward checking-ის დროის გასაუმჯობესებლად თავიდან მხოლოდ lcv დავამატე, ანუ ყოველი უჯრის შევსებამდე ჯერ ვითვლიდი თითოეული რამდენ დომეინში შლიდა მნიშვნელობებს და ამ რიცხვის ზრდადობით ვალაგებდი და ამ დალაგების მიხედვით ვცდილობდი ჩამესვა მნიშვნელობები. როცა მხოლოდ forward checking საშუალოდ **თითო** სუდოკუს 10-15წთ-ში ხსნიდა, forward checking + lcv საშუალოდ ნახევარ წუთს ანდომებდა, რაც ძალიან დიდი სხვაობაა, თუმცა ნახევარი წუთიც ბევრია. შემდეგ დავამატე mrv-ც, ანუ მანამდე თუ უბრალოდ მიმდევრობით გადავყვებოდი უჯრებს და ასე ვავსებდი, mrv-ს დამატების მერე ვნახულობდი თითოეულ შეუვსებელ უჯრას რამხელა დომეინი ჰქონდა და მათგან მინიმალურის მქონეს ვირჩევდი შემდეგ შესავსებად. ამან კიდევ უფრო გააუმჯობეს დრო, დაახლოებით 2-ჯერ.

თუმცა, რომ დავაკვირდი, lcv-ს დათვლას ყოველ ჯერზე თითქმის იმდენივე დრო სჭირდება, რამდენიც თვითონ დომეინების დააფდეითებას, იმის გამოკლებით, რომ დომეინებიდან არ შლის არაფერს, ხოლო mrv-ს ქონის ფონზე, მისი ეფექტი თითქმის ქრება, რადგან ძირითადად დომეინში ყველაზე ნაკლები მნიშვნელობის მქონე უჯრის დომეინი 3-ზე მეტი სიგრძის არ არის ხოლმე და ამ 3 მნიშვნელობიდან ყველაზე ნაკლებად შემზღუდავის არჩევა იმდენ მოგებას არ იძლევა, რომ მის მიერ დახარჯულ დროის რესურსად ღირდეს. ამ ლოგიკით გადავწყვიტე lcv წამეშალა და ისე მეცადა და გაამართლა, უკვე საშუალოდ 0,37 წამში ხსნიდა თითო სუდოკუს. თუმცა, ამ დროს ჯერ კიდევ ვაკოპირებდი დომეინებს ყოველ ჯერზე, რასაც ვიცოდი რომ დიდი რესურსი მიჰქონდა, ამიტომ შევცვალე ეს list-ში შეცვლილი დომეინების ინდექსების შენახვით შემდგომ აღსადგენად და ძალიან კარგი შედეგი მივიღე, მთელ მოცემული ფაილის სუდოკუებს საშუალოდ 17 წამში ხსნის, შესაბამისად ერთ სუდოკუს საშუალოდ 0,0168 წამში.

ცხადია, ეს ვარიანტი დავტოვე, რადგან ყველაზე სწრაფიც იყო და ყველაზე ლოგიკურიც. დროის მხრივ დიდ როლს თამაშობს სხვადასხვა ინფორმაციის შესანახად არჩეული მონაცემთა სტრუქტურებიც. სუდოკუების, დომეინების ან შეცვლილი დომეინების შესანახად სხვანაირი სტრუქტურები რომ ამერჩია, დრო შეიცვლებოდა ცხადია, მაგრამ მგონია, რომ ჩემი არჩეული სტრუქტურები შეძლებისდაგვარად ოპტიმალურია, განსაკუთრებით საუდოკუს ერთ განზომილებიან list-ში შენახვა, რადგან 9 ჩადგმულ ლისტს არ ვქმნი ზედმეტად.

| Algorithms and heuristics | Average time per sudoku* |
|---|---|
| Forward checking | 10-15mins |
| Forward checking + LCV | 30secs |
| Forward checking + LCV + MRV | 0,03secs |
| Forward checking + MRV | 0,0168secs |
| Arc consistency | 20-25mins |

* მხოლოდ forward checking-ის და arc consistency-ს შემთხვევაში იმდენ დროს უნდებოდა ზოგიერთს, რომ ფიზიკურად ვერ დაველოდებოდი სანამ მთელი ფაილის სუდოკუებს ამოხსნიდა (ეს იმიტომ, რომ ამ დროს ჯერ კიდევ ვაკოპირებდი დომეინებს, რაც ძალიან ანელებდა), ამიტომ დროის შეფასებები შედარებით უხეშია, ანუ ნაკლებ მონაცემზე დაყრდნობით არის გაკეთებული. ასევე arc consistency + MRV, arc consistency + LCV და arc consistency + MRV + LCV-ს შემთხვევაში დროებს არ ვაბეჭდინებდი, უკვე წაშლილი რომ მქონდა ესენი მაშინ დავიწყე და ის კოდები აღარ მქონდა (ამ ეტაპზე მხოლოდ ყველაზე სწრაფ ალგორითმს ვეძებდი და არ მეგონა ზუსტი დროები თუ გამომადგებოდა), უხეშ მიახლოებას ამათზეც გავაკეთებდი, თუმცა, გადავწყვიტე არ შემეტანა ცხრილში.