A COMPARATIVE STUDY ON THE PERFORMANCE OF SEARCH ALGORITHMS SOLVING FUTOSHIKI
CS7IS2 Project (2019-2020)

Aditya Vishnu, Alexandra Silva, Cian Conry, Patrick Prunty, Yash Pandey

adgarg@tcd.ie, asilva@tcd.ie, conryc@tcd.ie, pprunty@tcd.ie, pandeyy@tcd.ie

Abstract: Solving Futoshiki, a NP-Complete optimization problem using Generalized Backtracking, Forward Checking, AC-3, and Simulated Annealing. The study is aimed at finding the best algorithm in terms of least time consumption in solving Futoshiki and least number of moves in Futoshiki. The performance characteristics of each algorithm are studied by solving different difficulty levels of the puzzle. The comparative study reveals the superiority of the Forward Checking algorithm over the rest.

1. INTRODUCTION

A Constraint Satisfaction Problems (CSP) consists of decision variables with associated domains (possible values), constraints on the possible assignments of values to decision variables, and optionally an objective function [1]. CSPs could be anything from a theoretical problem such as map colouring to a real-world problem or a logic puzzle. A real-world example of a CSP would be a scheduling or planning problem; domains exist (possible values are hours of the day not spent sleeping) and there exist constraints (cannot schedule an activity when another is scheduled). Puzzles such as Sudoku and Futoshiki are some of the many possible logic CSPs. Logical puzzles have always been attractive to researchers in the field of artificial intelligence. This is due to the very idealised problem characteristics such as perfect information, deterministic actions, and a clear goal state.
Futoshiki is a Japanese logical puzzle shown as an n x n grid of cells. N refers to the dimension of the puzzle. To solve the puzzle correctly two separate conditions must be satisfied, the Latin square condition and the inequality condition. To satisfy the Latin square condition, each integer $\{1,2, \dots, n\}$ must appear exactly once in each row and column. The inequality condition means, the integers assigned to two adjacent cells who share an inequality sign must satisfy this inequality [2]. For example, if cell one must be less than cell two, and cell two is equal to three, cell one can only be equal to one or two. An example of a Futoshiki problem as well as its solution is shown in Figure 1.

Initial problem Solved problem

Figure 1: 5x5 Futoshiki puzzle example

The application of artificial intelligence algorithms to this simple problem is a sensible first investigation when analysing and comparing the performance of multiple algorithms. The fundamental ability to attain a solution to a simple problem and the efficiency (required computational complexity) required are all factors that may be easily determined for a simple problem before they are investigated for a more complex problem.

2. PROBLEM DEFINITION

Futoshiki may be modelled as a CSP with discrete variables (for a 5x5 dimension problem there are 25) and finite domains (every element may only contain a value from the set). The constraints on the possible values from the domain that each variable may take are as follows:

- Unique value constraint: The five variables that make up every row or column in the Futoshiki table may not use repeated values from the domain; each value in the domain may be used only once.
- Inequality constraint: There exists an inequality constraint that is unique to each puzzle which limits the possible values from the domain an element in the Futoshiki table may have relative to another. The inequality constraint consists of less than and greater than constraints.

The unique value constraint applies to all variables in the puzzle and remains the same for every puzzle. The inequality constraint is unique to every puzzle and will only affect specific variables. The domain for each variable remains the same (for a 5x5 dimension some value from 1 to 5) unless the variable is affected by any inequality constraint. The domain becomes smaller if the variable is affected by an inequality constraint. For example, if the variable is affected by a less than constraint the domain will become 1 to 4 for a 5x5 dimension puzzle. Inequality constraints are unique to each puzzle. Additionally, some puzzles have some predefined value for certain variables. This is less of a constraint and more of a feature that reduces the state space for the puzzle. This feature can both increase or decrease the difficulty of the puzzle.

The following features were considered when determining what type of algorithm to apply to the Futoshiki problem:

- The actions are deterministic (for this problem the actions would be assigning a value from the domain to a variable), which is to say that there is no probability or chance to consider when assigning a value to a variable.
- The constraints are hard constraints, there are no soft constraints or preferences that require ranking different possible solutions against each other.
- There is a clear goal state.

Due to these features, the chosen algorithms include simulated annealing, backtracking, forward checking, and AC-3.

3. RELATED WORK

Four different algorithms are investigated as possible approaches for solving the Futoshiki puzzle problem. The four algorithms to be investigated are as follows. Backtracking, Forward Checking augmented Backtracking, AC-3, and Simulated Annealing. The latter algorithm, simulated annealing, unlike the other algorithms, is not traditionally used in control satisfaction problems, and instead is used in optimization problems. This algorithm was thus used as an alternative approach for solving Futoshiki. A brief description of each algorithm along with their similarities and differences are presented here.

An example of backtracking and simulated annealing tested against a problem similar to Futoshiki is examined in [3], where each algorithm is used to solve a Sudoku puzzle. In this paper, the backtracking algorithm outperforms the simulated annealing algorithm, both in computation time, as well as in success rates for solving the puzzles; particularly at a higher puzzle difficulty. As well as this, the backtracking algorithm will always find a solution if one exists, while simulated annealing has no parameters to suggest that the algorithm has found a solution. Instead, it relies on an energy function that it attempts to maximize through stochastic hill climbing; wherein the solution can be (in the very unlikely case) realised in the early stages of the process, but unrecognized.

The improvement in efficiency due to the introduction of filtering through the use of the Forward Checking algorithm is demonstrated in [4]. Forward Checking serves to limit the explorable state-space, thus reducing the time required to find the solutions if any exist. This is at the cost of extra computation per move.

AC-3 is a fast and easily implemented algorithm for ensuring arc consistency when compared to other arc consistency algorithms. It is a more effective and efficient algorithm than AC-1 and AC-2 [5], while according

to [6] it is generally superior to newer algorithms such as AC-4. AC-3 is often a more efficient algorithm than AC-4 due to the costly constraint tests of AC-4 [5]. One primary feature of AC-4 is that it takes certain measures to avoid a worst case performance scenario [5], which is often unnecessary for many problems [6].
AC-3 is similar to the filtering performed by the Forward Checking algorithm. Both involve checking the values within the domain of some variable that satisfies its constraints. While AC-3 operates before the backtracking is applied, Forward Checking acts while backtracking is solving the problem. Though AC-3 can be applied in a manner much more similar to Forward Checking as a Maintaining Arc Consistency algorithm [7][8].
An exploration into the benefits of AC-3 as a complete preprocessing method before the application of a solver is described in [9], where prior to its introduction to a CSP solving algorithm there was an issue of solution rate, a problem alleviated by the hybrid implementation of AC-3.
Simulated annealing does not always find the solution, and similarly cannot be used to determine if a solution even exists. This issue is mentioned in paper [10]; where unlike backtracking, simulated annealing is unable to guarantee a solution for the more difficult problems while also consistently taking more computation time to solve the problem..
An investigation into many of the algorithms in this paper is outlined in [11], where these algorithms are applied to an N-Queen problem, which is a typical CSP benchmark problem. Backtracking is described as a naive technique with a time to compute the solution that increases drastically for problems with a larger N. Filtering techniques such as Forward Checking and AC-3 were effective at improving the solution computation time due to their ability to effectively reduce the amount of time spent in dead ends.

4. SIMULATED ANNEALING ALGORITHM

Simulated annealing is a higher-level metaheuristic procedure which combines the stochastic process of a random walk with hill climbing.
A hill-climbing algorithm that never makes “downhill” moves toward states with a lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum. In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient. Therefore, it seems reasonable to try to combine hill-climbing with a random walk in some way that yields both efficiency and quasi-completeness. Simulated annealing is such an algorithm.
Imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface. If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum. The trick is to shake just hard enough to bounce the ball out of local min-ima but not hard enough to dislodge it from the global minimum. The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature). [12]
This notion of slow cooling implemented in the Simulated annealing algorithm is interpreted as a slow decrease in the probability of accepting worse solutions as the solution space is explored. Accepting worse solutions is a fundamental property of metaheuristics because it allows for a more extensive search for the global optimal solution.
In general, the Simulated Annealing algorithms work as follows. The temperature progressively decreases from an initial positive value to zero. At each time step, the algorithm alters the current configuration of the system slightly, measures the quality of the new configuration, and moves to it according to the temperature-dependent probabilities of selecting better or worse configurations, which during the search respectively remain at 1 (or positive) and decrease towards zero.

4.1. APPLICATION

The simulated annealing process is traditionally used in optimization problems, however, the metaheuristics of the process may be applied to constraint satisfaction problems, such as in puzzle games, where constraints are initially specified. In the case of Futoshiki, there seems to be an infinite number of possible configurations where a human, or a computer, can get stuck in a local maxima; for example, when inequality constraints are completely satisfied but the row and column constraints are not, or vice versa. This is when it becomes abundantly necessary to accept downhill movements; to abandon inequality constraint satisfaction, and thus

move out of local maxima, in the hope that a worse configuration will later lead to a better configuration, wherein inequality constraints are satisfied but so too are the row and column constraints.

4.2. IMPLEMENTATION

The pseudocode for the Simulated Annealing algorithm, as implemented inside the program, is as shown in Figure 2.

Algorithm 1 SIMULATED-ANNEALING

```
1:  C = C_int                                  ▷ Begin with initial configuration
2:  T large                                    ▷ Initialize maximum temperature
3:  r small                                    ▷ Initialize cooling rate
4:  while T > 0 do
5:      E_c = E(c)                             ▷ Compute energy of current configuration
6:      N = N(c)                               ▷ Alter the current configuration slightly
7:      E_N = E(N)                             ▷ Compute energy of new (altered) configuration
8:      ΔE = E_N − E_c                         ▷ Compute the change in energy
9:      if ΔE > 0 then                         ▷ Accept uphill move unconditionally
10:         C = N                              ▷ Make uphill configuration the new current configuration
11:     else if e^(ΔE/T) > rand(0, 1) then     ▷ Accept downhill move with probability
12:         C = N                              ▷ Make downhill configuration the new current configuration
13:
14:     T = T * (1 − r)                        ▷ Cool down the system
```

Figure 2: Simulated annealing algorithm pseudocode (image processed with LaTeX).

The algorithm begins with an initial configuration, $C = C_{int}$, that is initialized inside the initial_configuration() function. This function initializes the current configuration of the board. Any values that are initially specified at the beginning of the game are initialized here, whereas all other values are initialized to the value one. Two important variables in the simulated annealing process are then initialized, the temperature of the system T and its cooling rate r. We begin with an initial high temperature, $T = 100$, because we intend to slowly cool the system using the cooling rate variable $r = 0.000002$.

The algorithm then computes the energy of the current configuration, $E = E(c)$, using the compute_energy() function. This function is the key to the simulated annealing process. The simulated annealing process will only alter the current configuration if the new configuration is better than the old one, or else, it will accept the bad configuration with some probability. Thus, reward increments are used to represent the energy of the system, and furthermore, reveal how close to the solution (the global maximum energy) the current configuration is.

After computing the energy of the current configuration, it then slightly alters the current configuration of the system, $N = next(c)$, using the alter_configuration() function. The alter_configuration() function uses the std:: shuffle function from the main to generate two random integers, i and j, in the range `[0,4]` and uses them as indexers on each iteration to alter one of the elements on the Futoshiki board: data[i][j] = some new random value in the range `[1,5]`, and thus generate a new configuration of the board.

The algorithm then computes, $\Delta E = E_N - E_C$, the change in energy of the new configuration compared to the current one. This is equivalent to asking, "is the newly generated configuration closer to the solution?" If the change in energy is positive, i.e $\Delta E > 0$, then the change in energy is a good one, and the algorithm accepts the new configuration unconditionally (since it is 'closer' to a solution). The algorithm then makes the new configuration, N, the current configuration, C, $C = N$. In other words, the algorithm accepts 'uphill' moves unconditionally.

However, unlike a hill-climbing algorithm, the simulated annealing algorithm also accepts downhill movements with a certain probability. If the probability parameter, $e^{\Delta E/T}$, is greater than some random number between 0 and 1, i.e $e^{\Delta E/T} > rand(0, 1)$, then we accept the downhill movement. Due to the nature of this probability parameter, $e^{\Delta E/T}$, the algorithm accepts a large number of downhill movements when the temperature T is large, and as the temperature T cools (decreases), the algorithm begins to accept only uphill movements. The probability of a downhill move is shown in Figure 3.

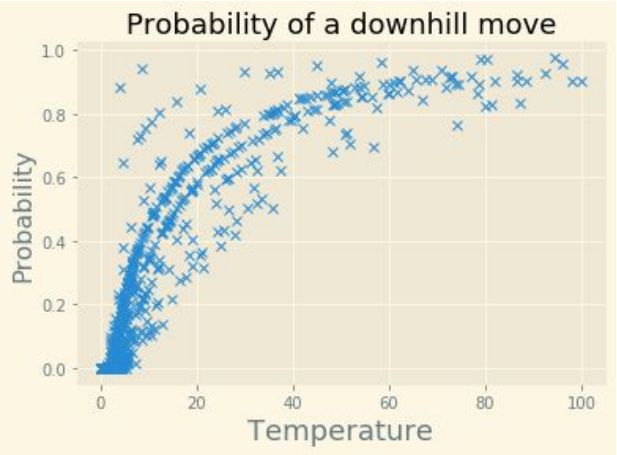


Figure 3: The probability the simulated annealing algorithm accepts a downhill movement, as the temperature decreases (look at this graph from right to left).

5. BACKTRACKING ALGORITHM

Backtracking is one of many search algorithms used to solve Constraint Satisfaction Problems. The generic backtracking algorithm was first described more than a century ago and since then, has been rediscovered many times. Backtracking algorithms are generally complete algorithms (guarantee that a solution will be found) [12]. The algorithm is a general uninformed search algorithm for solving problems recursively. Backtracking combines Depth First Search, variable ordering, and failure on the violation.

Depth First Search is a recursive algorithm that starts at the root node and expands each node as far deep as possible before changing branches. Backtracking can be used on any type of structure, while Depth First Search is limited to working on tree structures. Backtracking Search also implements two improvements to Depth First Search. Firstly, variable ordering. Assignments of variables are commutative, otherwise known as fixed. At each step of the algorithm, we consider a single variable. Secondly, fail on the violation. Check the constraints of the problem as you go. Rather than considering all values, only consider values that do not conflict with any previous assignments.

The Backtracking search algorithm selects unassigned variables, one by one, and assigns them each a value. The value assigned is based on the given constraints. If no value can be correctly assigned, due to the constraints, the algorithm backtracks to the previous variable and assigns a different value.

5.1 IMPLEMENTATION

The pseudocode for the basic Backtracking search algorithm can be seen in Figure 4.

```
BT (V, I)
Begin
    If V = ∅ then
        I is a solution
    Else
        Let x ∈ V
        For each v ∈ D_x do
            If I ∪ {(x, v)} is consistent then
                BT (V- {x}, I ∈ {(x, v)})
            End if
        End for
    End if
End
```

Figure 4: Backtracking Algorithm Pseudocode [5]

The algorithm's principle consists of starting from an empty instantiation and performing a tree traversal on the set of variables. The instantiation of a variable by a value from its domain triggers a consistency verification procedure for every constraint that involves the variable [5].

The implemented algorithm retrieves the next empty variable. It then loops through every possible value and checks if it can be assigned to the variable based on the variables constraints. In this case, the algorithm checks if there are any duplicate values in the row or column of the variable. It then checks if the variable is affected by any more than or less than constraints. If the value passes all constraints it is assigned to the variable and the algorithm moves on to the next empty variable. If there is no possible value for the variable the algorithm backtracks and reassigns the previous variable.

6. AC-3 ALGORITHM

In the standard state-space search, only one thing can be achieved by an algorithm: search. However, while dealing with CSPs there is a choice, an algorithm can search or perform a specific form of inference called constraint propagation that is using constraints to reduce the number of legal values for a variable, which in effect will reduce the legal values for another variable, and so on. Constraint propagation can be combined with the search or can be performed as a preprocessing phase before the search begins. Often this preprocessing might solve the entire problem as well.[13]
A variable in a CSP is arc-consistent if any value in its scope meets the binary constraints of the variable. More formally, Xi is arc-consistent with respect to another variable Xj if for every value in the current domain Di there is some value in the domain Dj that satisfies the binary constraint on the arc (Xi, Xj). A network is arc consistent if every variable is arc consistent with every other variable.[5]
AC-3 is the most popular and most used Arc Consistency algorithm. Considering a complete constraints graph, for O(n^2d) arcs, the temporal complexity is O(n^2d^3) and the spatial complexity is O(n^2).

6.1. IMPLEMENTATION

The pseudocode for the AC-3 algorithm can be seen in Figure 5:

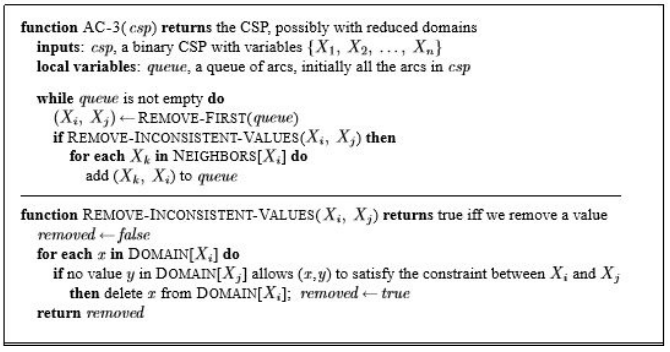


Figure 5: AC-3 Algorithm Pseudocode [13]

This algorithm uses a single queue structure. The queue includes all the arcs in the CSP. AC-3, then pops up a random arc (Xi, X j) from the queue and renders Xi X-consistent with Xj. If this leaves Di unaffected, then the algorithm just advances to the next arc. But if this affects Di, that is if it makes the domain smaller, then we add all the arcs (Xk, Xi) where Xk is Xi's neighbour, to the queue. We need to do so as the change in Di may permit more reductions in Dk domains, even though we have previously considered Xk. If Di is revised down to nothing, then we know that the whole CSP has no consistent solution, and AC-3 can return failure immediately. Otherwise, we keep searching, attempting to eliminate values from variable domains until there are no more arcs in the list. On this point, we're left with a CSP that is similar to the initial CSP, they all have the same solutions — but in most instances, the arc-consistent CSP would be easier to evaluate as the variables have smaller domains [13].

7. FORWARD CHECKING ALGORITHM

The conventional application of the backtracking algorithm does not make use of any heuristics to execute pruning operations in a tree. When allocating a variable, the value and constraints imposed on that variable may be employed to prune the set of the future variables. However, this pruning is not done in plain backtracking search, which makes it not really efficient or intelligent. It does not make use of the information provided in the constraint problem and the allocated variables

In the improved backtracking, we make use of Forward Checking (FC) as a pruning technique. It does an assignment of a variable which has no legitimate domain values in a tree with various possibilities. This validation of variables is done for every assignment in the problem statement. If the domain becomes null for any of the future variables, then the value being under selection is discarded and backtrack is performed where the next alternative is sought. While doing the backtracking, it is also important to make sure that all the values that were discarded because of the trial assignment are undone, which is one of the computation overhead in FC. This entire process is implemented out until a solution is found, or the algorithm has tested every possibility using pruning (under a defined set)

7.1. IMPLEMENTATION

The pseudocode for the algorithm is depicted in Figure 6.

```
procedure SELECT-VALUE-FORWARD-CHECKING
    while D_i' is not empty
        select an arbitrary element a ∈ D_i' and remove a from D_i'
        for all k, i<k ≤ n
            for all values b in D_k'
                if not CONSISTENT(ā_{i-1}, x_i = a, x_k = b)
                    remove b from D_k'
            end for
            if D_k' is empty                (x_i = a leads to a dead-end don't select a)
            reset each D_k', i < k ≤ n to value before a was selected
            else
            return a
    end while
    return null                             (no consistent value)
end procedure
```

Figure 6: Forward Checking Pseudocode [14]

The algorithm begins, where variables from x_1 to x_{i-1} have been initialized. We define x_i as the current variable and a_i as the temporary solution of the problem. The algorithm discards all the values in the domain of x that have inconsistency with a_i for any of the constraints. Also, If the domain becomes null for any of the future variables x, then the temporary solution a_i is discarded.

The complexity of the given algorithm is O(*nd*), wherein *n* represents the number of constraints and *d* is the degree of cardinality for the largest domain.

Disadvantage: The forward search does more calculation per allocation than the traditional backtracking algorithm.

Advantage: It saves time by finding values and cases which are inconsistent with the constraints in the problem.

8. EXPERIMENTAL RESULTS

The Simulated Annealing process was successful in solving both the ‘easy’ and ‘extreme’ difficulty 5x5 futoshiki puzzles. However, due to the stochastic nature of the simulated annealing processes, it does not always converge to the solution (the global maximum).

The program Futoshiki, which finds the solution for the 'easy' difficulty Futoshiki game board 95% of the time, has a run time of 1.170598 seconds and finds the correct solution after an average of 1754265 moves. The solution can be seen in Appendix B.
The initial configuration for this puzzle has the usual 5 row and 5 column constraints, and 9 inequality constraints that need to be satisfied. Thus, from the compute_energy() function, the maximum global energy is at 178.00. The stochastic global ascent to this maximum global energy is shown in Figure 7.

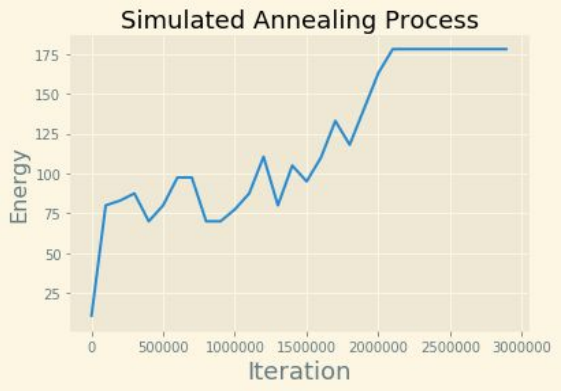


Figure 7: Stochastic ascent to the global maximum for 'easy' difficulty.

The program futoshiki2, which finds the solution for the 'extreme' difficulty Futoshiki game board 65% of the time, has a run time of 0.868288 seconds and finds the correct solution after an average of 1712334 moves. The solution can be seen in Appendix C.
This configuration for this puzzle already has three elements of the board initially specified, the usual 5 row and 5 column constraints, as well as 9 inequality constraints that need to be satisfied. Thus, from the compute_energy() function, the maximum global energy is at 165.5. The stochastic global ascent to this maximum global energy is shown in Figure 8.

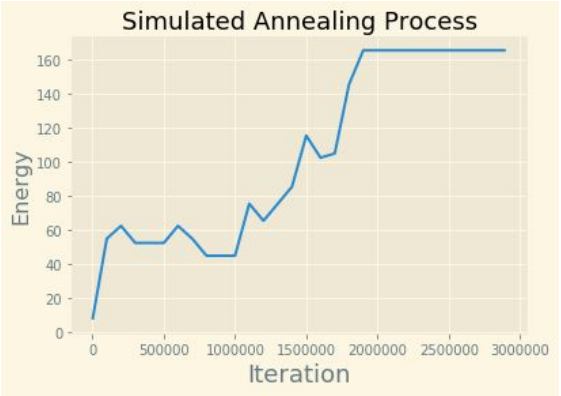


Figure 8: Stochastic ascent to a global maximum for 'extreme' difficulty.

Whilst the simulated annealing algorithm, through the clever use of the rewards system in the compute_energy() function, succeeded in being applied to this control satisfaction problem, it lacks the search pruning present in the other algorithms to be considered complete.
AC-3 can be combined with the search algorithms which in our case are backtracking and forward checking or can be performed as a preprocessing phase before the implementation of search algorithms. AC-3 effectively mitigated any inconsistencies that were apparent in each of the arcs under evaluation in our four test cases. As shown in the results, each variable was arc consistent. Often this preprocessing might solve the entire problem as well, however in the test cases under consideration it failed to solve any problem completely. In the results shown in Appendix A, (A, B, C ..) represents the rows of the game and (1,2,3 ..) represents the columns. The

results show that the algorithm was successful in maintaining arc consistency[Appendix A]. However, the algorithm was unable to solve any puzzle making it unsuccessful in that respect.
The backtracking algorithm was successful in solving every size of the Futoshiki puzzles. For every puzzle, the algorithm converges to a solution 100% of the time proving the algorithm is complete. Similarly, the improved backtracking algorithm which uses the Forward Checking was also able to find a solution for every Futoshiki puzzle. As seen in Figures 9 and 10, both the Backtracking algorithm and the Forward Checking algorithm perform similarly in respect to the number of moves needed to solve and the time taken to solve until the dimensions of the puzzle increase. As the dimensions increase it becomes clearer that Forward Checking is the best performing algorithm.

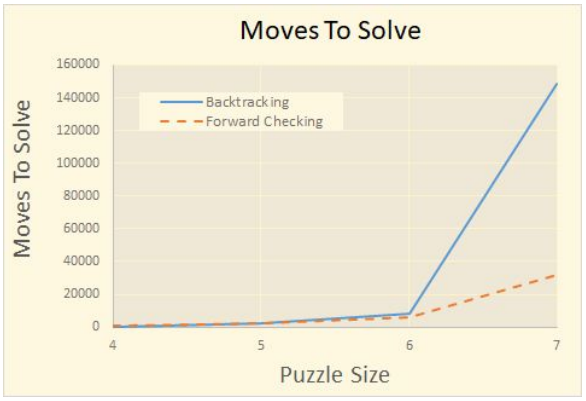


Figure 9: Moves to Solve vs Puzzle Size

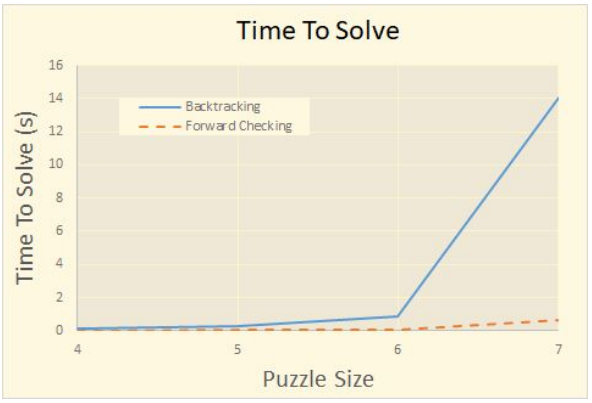


Figure 10: Time to Solve vs Puzzle Size

9. CONCLUSION

In this paper, Futoshiki acts as a simple logic puzzle for testing the following artificial intelligence algorithms: Backtracking, Forward Checking, AC-3, and Simulated Annealing. From the results presented, Forward Checking is the optimal algorithm. This is due to the following characteristics: A guaranteed solution, the fastest solution time, the least number of moves to solve, and solution time and number of moves to solve remain the most consistent as complexity increases. By comparison, backtracking on its own shows significant increases in solution time for problems with greater complexity.
While AC-3 is an effective method for maintaining arc consistency, which itself may be adequate to find the solution to the problem; it is principally intended as a preprocessing step for another CSP solving algorithm. Due to the algorithm's unsuitability to solving CSPs on its own, it is not the proposed algorithm.
The simulated annealing algorithm, on the other hand, despite being traditionally used in optimization problems, successfully solved both of the 5x5 futoshiki puzzle games, and thus, succeeded in application to a control satisfaction problem, but suffered from a large computation time, especially when compared to that of the backtracking based solvers. This, in addition to the algorithm's overreliance on 'good' pseudo-random number generation, meant that it would find the solution to the 5x5 'extreme' difficulty only 65% of the time. Making the algorithm a suboptimal solution to the control satisfaction problem of solving futoshiki, especially when compared to the backtracking algorithms, which finds the solution 100% of the time.

REFERENCES

[1] Salamon AZ. Transformations of representation in constraint satisfaction. Constraints. 2015;20(4):500–1.
[2] Haraguchi K. The Number of Inequality Signs in the Design of Futoshiki Puzzle. J Inf Process. 2013;
[3] Chi EC, Lange K. Techniques for Solving Sudoku Puzzles. 2012.
[4] Zhao C, Cui Y. An Improved Algorithm of CSP. Procedia Eng. 2012
[5] Ghedira K. Constraint Satisfaction Problems: CSP Formalisms and Techniques. Wiley; 2013.
[6] Wallace RJ. Why AC-3 is almost always better than AC-4 for establishing arc consistency in CSPs. In: IJCAI. 1993. p. 239–45.
[7] Régin J-C. Maintaining Arc Consistency Algorithms During the Search Without Additional Space Cost. In 2005. p. 520–33.
[8] Bacchus F, Grove A. Looking forward in constraint satisfaction algorithms. Unpubl Manuscr. 1999;
[9] Soto R, Crawford B, Galleguillos C, Monfroy E, Paredes F. A hybrid AC3-tabu search algorithm for solving Sudoku puzzles. Expert Syst Appl. 2013;40(15):5817–21.
[10] Chi EC, Lange K. Techniques for Solving Sudoku Puzzles. 2012.
[11] Ayub MA, Kalpoma KA, Proma HT, Kabir SM, Chowdhury RIH. Exhaustive study of essential constraint satisfaction problem techniques based on N-Queens problem. In: 2017 20th International Conference of Computer and Information Technology (ICCIT). 2017. p. 1–6.
[12] van Beek P. Backtracking Search Algorithms. Elsevier. 2006
[13] Russell, Stuart J. (Stuart Jonathan). Artificial Intelligence : a Modern Approach. Upper Saddle River, N.J. :Prentice Hall, 2010
[14] Meisels A. Distributed Search by Constrained Agents. London: Springer-Verlag London Limited, 2008.

Appendix

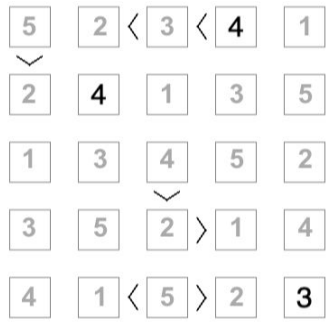
Appendix A - Results of AC-3

```
Test-case 3

AC-3 finished, executed for 936 times
Time: 0.0086 seconds
A1 { [1, 2, 3, 4, 5] }, A2 { [1, 2, 3, 4, 5, 6] }, A3 { [1, 2, 3, 4, 5, 6] }, A4 { [1, 2, 3] }, A5 { [3,
4] }, A6 { [4, 5] },
B1 { [2, 3, 4, 5, 6] }, B2 { [1, 2, 3] }, B3 { [2, 3, 4] }, B4 { [1, 2] }, B5 { [2, 3] }, B6 { [5, 6] },
C1 { [1, 2, 3, 4, 5, 6] }, C2 { [1, 2, 3, 4, 5, 6] }, C3 { [3, 4, 5] }, C4 { [4, 5, 6] }, C5 { [1, 2, 3,
4, 5, 6] }, C6 { [1, 2, 3, 4, 5, 6] },
D1 { [3, 4, 5, 6] }, D2 { [2, 3, 4, 5] }, D3 { [1, 2, 3, 4] }, D4 { [1, 2, 3, 4, 5] }, D5 { [2, 3, 4, 5,
6] }, D6 { [3, 4, 5, 6] },
E1 { [1, 2, 3, 4, 5, 6] }, E2 { [1, 2, 3, 4, 5] }, E3 { [3, 4, 5, 6] }, E4 { [1, 2, 3, 4, 5, 6] }, E5 {
[1, 2, 3, 4, 5, 6] }, E6 { [2, 3, 4, 5] },
F1 { [1, 2, 3, 4, 5, 6] }, F2 { [2, 3, 4, 5, 6] }, F3 { [2, 3, 4, 5] }, F4 { [1, 2, 3, 4] }, F5 { [1, 2,
3, 4, 5, 6] }, F6 { [1, 2, 3, 4] },
----------------X----------------
Test-case 4

AC-3 finished, executed for 1352 times
Time: 0.0130 seconds
A1 { [1, 2, 3, 4, 5] }, A2 { [1, 2, 3, 4, 5, 6, 7] }, A3 { [1, 2, 3, 4, 5, 6] }, A4 { [2, 3, 4, 5, 6, 7]
}, A5 { [1, 2] }, A6 { [2, 3] }, A7 { [6, 7] },
B1 { [2, 3, 5, 6] }, B2 { [1, 2, 3, 5, 6, 7] }, B3 { [2, 3, 5, 6, 7] }, B4 { [1, 2, 3, 5, 6] }, B5 { [5,
6] }, B6 { [4] }, B7 { [5, 6] },
C1 { [3, 4, 5, 6, 7] }, C2 { [2, 3, 4, 5, 6] }, C3 { [1, 2, 3, 4, 5, 6, 7] }, C4 { [2, 3, 4, 5, 6, 7] }, C
5 { [6, 7] }, C6 { [1, 2, 3, 5, 6, 7] }, C7 { [6, 7] },
D1 { [1, 2, 3, 4, 5, 6, 7] }, D2 { [1, 2, 3, 4, 5] }, D3 { [4, 5, 6, 7] }, D4 { [1, 2, 3, 4, 5, 6, 7] }, D
5 { [1, 2, 3, 4, 5] }, D6 { [2, 3, 5, 6] }, D7 { [3, 4, 5, 6, 7] },
E1 { [1, 2, 3, 4, 5, 6, 7] }, E2 { [1, 2, 3, 4, 5, 6, 7] }, E3 { [3, 4, 5, 6] }, E4 { [1, 2, 3, 4, 5, 6,
7] }, E5 { [1, 2, 3, 4, 5, 6, 7] }, E6 { [1, 2, 3, 5, 6, 7] }, E7 { [1, 2, 3, 4, 5, 6, 7] },
F1 { [2, 3, 4, 5, 6, 7] }, F2 { [1, 2, 3, 4, 5, 6, 7] }, F3 { [2, 3, 4, 5] }, F4 { [1, 2, 3, 4, 5, 6, 7]
}, F5 { [2, 3, 4, 5, 6, 7] }, F6 { [1, 2, 3, 5, 6, 7] }, F7 { [2, 3, 4, 5, 6, 7] },
G1 { [1, 2, 3, 4, 5, 6] }, G2 { [1, 2, 3, 4, 5, 6, 7] }, G3 { [1, 2, 3, 4] }, G4 { [1, 2, 3, 4, 5, 6, 7]
}, G5 { [1, 2, 3, 4, 5, 6] }, G6 { [1, 2, 3, 5, 6, 7] }, G7 { [1, 2, 3, 4, 5, 6] },
----------------X----------------
```

Appendix B - Solved puzzle by simulated annealing (easy)



Appendix C - Solved puzzle by simulated annealing (hard)