Curtis Fedorko, Riley Olds, Eric Wnorowski

Professor Hunsberger

CMPU365: Artificial Intelligence

9 May 2023

Final Project: Constraint Satisfaction Problem - Sudoku

Introduction

Modern sudoku first began to rise in popularity in Japan during the late 20th century and eventually, Wayne Gould would take initiative to spread the puzzle to the rest of the world. Wayne first approached *The Times* in London with his computer software that could rapidly produce unique puzzles. The rest is history as we know it, sudoku has become one of the most popular puzzles in the world and is a staple game of newspapers, computers, online games, etc. Sudoku is also one of the more popular examples of a Constraint Satisfaction Problem (CSP). There are three key components to a CSP, the variables, the domains, and the constraints. For each problem, there is a set number of variables that have specific domains to which their values can be assigned and the constraints which are a relation over a subset of the variables that restrict the combinations of values. The reason Sudoku is commonly studied in Computer Science and AI is because of the various techniques that can be used to implement a sudoku solver.

1	5	7	6	4			8	
	4							
	3	2	9			1	4	
7		4	1		5	2		
2			8	6			7	4
				7				1
	8			2	1			
			3		4		1	9
			5		6	8	2	

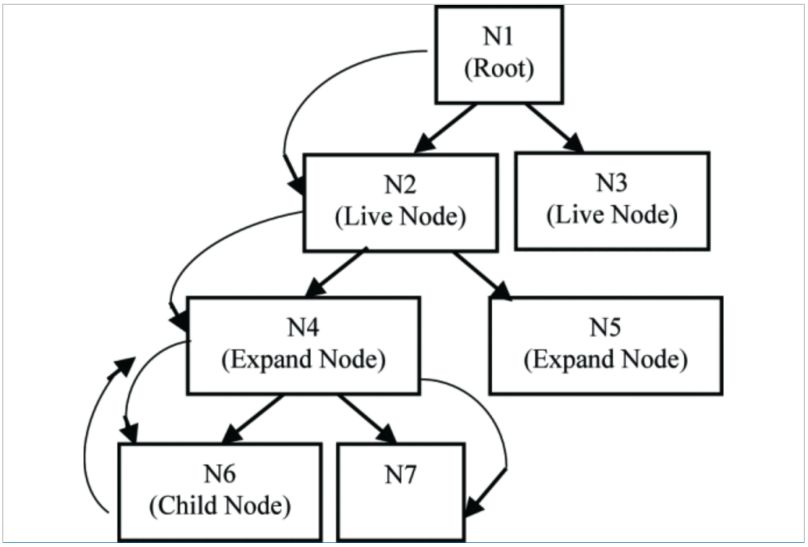
Fig. 1.1: Sample Puzzle

For the final project, these techniques are implemented and tested to be able to compare, discover, and learn the details of each method. Typically sudoku puzzles can vary in size (variable

n) and difficulty based on the number of cells. In our implementation, we look to use "nines" sudoku puzzles (*n-sized* puzzles) which means it is an NP-complete CSP. The variable *n* refers to the number of rows, columns, and the number of cells within each box. These are the aforementioned set of constraints for this particular CSP. While the domain is the possible values of zero to *n* and the variables are all the initially unfilled cells. The clearly defined constraints of sudoku make it easy to heuristically compare two sudoku game states. This is one commonality amongst some of the algorithms we will discuss, many of them will use all or some of these three constraints to help guide the search. In the case of nines puzzles the state space can get increasingly large based on the difficulty of the puzzle. Therefore, any sudoku solver must be successful in operating within the constraints and cutting off portions of the search tree that are not valuable. If part of the search can not be pruned then the algorithms may take a sufficiently long time to come to a solution. For this project one of the faster sudoku-solving algorithms that utilizes forward checking was already implemented by Professor Hunsberger. Therefore, it seemed important to understand how it was more efficient than other algorithms and why.

Backtracking

One of the most traditional implementations of a sudoku solver is backtracking because it utilizes the common depth-first search algorithm. See the search tree outlined below (Hasanah 2013). The main strategy of the algorithm is to continually try new moves until a variable is left without a domain. The basic



version of this algorithm does not implement consistency or forward checking. The algorithm only stops when it chooses an unfilled cell and there are no domains left, it will not check this in advance like other algorithms might. This results in the basic backtracking to be a particularly slow sudoku solver for large state spaces, although it is guaranteed to find a solution if there is one (it will try every possible number in every location). In the case of a puzzle being small or having a few number of unfilled cells then backtracking will be particularly efficient, but it can not scale with the game state as well as other algorithms.

There are a number of ways that backtracking can be improved. For example, the pseudocode below from the textbook demonstrates the variety of ways backtracking can be improved (Ernest 2010). The *SELECT-UNASSIGNED-VARIABLE()* and *INFERENCE()* functions can be used to implement heuristic methodologies like Minimum Remaining Values, Degree Heuristic, Least Constraining Value, Consistency, and Constraint Propagation. Many of these were discussed in class during the review of CSP search algorithms, so for the project, it was interesting to see the other possible sudoku-solving algorithms and how they utilize these different techniques to improve the algorithms. Before coding each into our lisp program it was necessary to walk/talk through each algorithm to understand the algorithm. When doing this it was clear the various heuristics and techniques by different algorithms often overlap, and the Professor's solution included a number of these

```
function BACKTRACKING-SEARCH(csp) returns a solution, or failure
  return BACKTRACK({ }, csp)

function BACKTRACK(assignment, csp) returns a solution, or failure
  if assignment is complete then return assignment
  var ← SELECT-UNASSIGNED-VARIABLE(csp)
  for each value in ORDER-DOMAIN-VALUES(var, assignment, csp) do
    if value is consistent with assignment then
      add {var = value} to assignment
      inferences ← INFERENCE(csp, var, value)
      if inferences ≠ failure then
        add inferences to assignment
        result ← BACKTRACK(assignment, csp)
        if result ≠ failure then
          return result
    remove {var = value} and inferences from assignment
  return failure
```

techniques in the execution of forward checking. The goal of this project is to gain a better understanding of the value of these optimizations and learn how to implement them in different ways to efficiently solve sudoku.

We implemented a basic version of backtracking to compare against the given implementation that included optimizations to increase efficiency (see code to the right). This method ran at $O(9^m)$ where m is the number of unfilled slots. It was very quick for solving small easy puzzles but very slow for solving larger harder puzzles.

```
;;; BACK-TRACK
;; --------------------
;; INPUT: G, A SUDOKU GAME
;; OUTPUT: A SOLVED SUDOKU GAME
;; SIDE EFFECT: PRINTS NUMBERS OF STEPS TAKEN TO SOLVE GAME

(defun back-track (g)
  ;; find an unfilled cell
  (let ((idx (has-empty g)))
    (cond
     ;; no empty slots.. we have valid sudoku
     ;; return true
     ((not idx)
      (format t "~A" g)
      (return-from back-track T))
     (t
      ;; try values 1 - 9 in the empty slot
      (dotimes (j 9)
        ;; if the value satifies constraints
        (if (check-constraints (do-move (copy-game g) idx (+ 1 j)))
            ;; move is valid and do the move
            (progn
              (setf g (do-move g idx (+ 1 j)))
              (if (back-track g)
                  (return-from back-track T)))
          ;; move is invalid.. try next value
          ;; set the slot back to being empty
          (setf (aref (game-slots g) idx) '_)))
      ;; tried all possible values and none worked
      ;; return nil
      (return-from back-track nil)))))
```

Simulated Annealing

Simulated Annealing is a stochastic search method that generates random neighboring game states and accepts them probabilistically. For the purposes of a sudoku puzzle, we start with a puzzle *P*, from which we generate an initial solution state *S*. To do this, we fill in each of the 9 3x3 constraint squares that make up the sudoku board with the numbers 1 to 9. The *fill-boxes* function takes in an initial puzzle and populates all of the boxes in it in this way. This initial filled board serves as our starting solution. Along with the starting solution, we provide an initial temperature of 1.0. We tried to calculate temperature using the *start-temp* function, which runs 200 calculations of random starting solutions and returns the standard deviation, providing a temperature relative to the problem at hand, however, we received better search results with a lower starting temp. When we initialize the *sim-annealing* function we provide it with the initial

puzzle and a step value. This specifies the number of temperature steps we are going to take; the intervals between geometrically decreasing the temperature according to the *cooling* function. This cooling rate greatly affects the breadth of the search and must be high enough to allow the search to avoid local minima. During each temperature step, we do a number of searches according to the number of initial blanks there are in the problem, hence relative to the search area.

The *sim-annealing* function generates a candidate solution, then does a number of searches at each temperature, passing the current state to the *neighbor* function along with the list of starting moves of the puzzle, which we retrieve from *collect-forced-moves*. The *neighbor* function selects a random index using *gen-idx*, which checks that it is not one of the forced slots. With our selected index, we pick another index within the same box using *pick-swap*, which we assure isn't a forced slot either and isn't the same as our selected index. We swap the values of the two slots in order to generate a neighboring state. We then calculate the cost of the two states evaluated according to *cost-func*, which sums the number of broken row and column constraints. These costs and the temperature are then passed to the *probability* function. The probability function returns a probability of acceptance equal to $exp(-\ (d\ -\ cost/temp))$ where d-cost is $newcost\ -\ currcost$. The new state is accepted automatically if it has a lower cost than the current state, and, even if it is worse, is accepted according to the result of the *probability* function (see code on the right within the loop of simulated annealing algorithm). If the new cost is better, it updates the low cost. In order to avoid being

```
;; create a neighbour and accept it according to probability
(let* ((nb (neighbour board start-moves lt?))
       (cost (cost-func board))
       (n-cost (cost-func nb))
       (p (probability cost n-cost temp)))
  (format t "$: ~A, N$: ~A, P: ~B~%" cost n-cost p)
  (format t "S: ~A, CL: ~A/~A, L$-c: ~A~%" i j chainl l-count)

  ;; p is accepted, update the board and low-cost/l-count if n-cost is lower
  (when (<= (random 1.0) p)
    (setf board nb)
    (when (< n-cost low-cost)
      (setf low-cost n-cost)
      (setf l-count 0))))))
```

locked into low-cost paths, we count the number of steps done at a given low cost. When we reach 20, we conduct reheating in which we reset the temperature and reinitialize the game state, effectively starting the search over with a reset game state. At the end of each step, the *cooling* function is run. This cycle is repeated for the number of specified steps until either a solution is found, or the search is exhausted.

We tested a number of different starting temperatures and cooling rates in conjunction with varied step counts and step lengths. Starting with a high initial temperature allows the search to get far from its initial state which helps us avoid local minima, however, too high a temperature will take forever to exploit the problem and find good results. We settled on an initial temperature of 1.0 and a cooling rate of 997/1000. This rate allowed for us to maintain a solid temperature across the whole search, helping us search wider when we got closer to the goal. We settled on a step length of $2 * count - empty(g)$, which gave us enough searches in a short amount of time at each temperature step. Because of our reheating procedure, we can ask our simulated annealing program to go for any number of steps. If we get stuck in a local minima it will reheat temperature and reinitialize the board, allowing us to restart the search.

Hill Climbing

One of the other common sudoku-solving algorithms that is often compared to backtracking is the hill climbing algorithm. The idea is to continually build upon changes to the unfilled cells in order to lower the number of constraints broken. As with simulated annealing, this

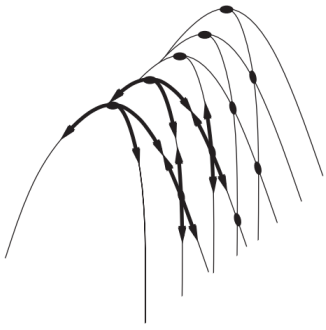


Figure 4.4 Illustration of why ridges cause difficulties for hill climbing. The grid of states (dark circles) is superimposed on a ridge rising from left to right, creating a sequence of local maxima that are not directly connected to each other. From each local maximum, all the available actions point downhill.

algorithm will take the board and fill in domain-specific values for each non-filled cell. This means any value that is in the cell domain is based upon the forced moves. This is not making the board consistent because when one value is picked for a cell it will not affect the other domains for constraint-related cells. The general hill climbing algorithm performs one valid move, in this case changing the value of a non-forced move cell, and then comparing the game states based on the constraint cost function. If the new game state is better than the previous one, it becomes the current game state and the algorithm continues from there. One issue that arises from this technique can be seen on the previous page (Ernest 2010). Hill climbing has a tendency to have search reach a local maximum that only allows moves from that game state to be worse, which means the algorithm can stall. Random restarts have been found to solve this issue by triggering a random restart in the search after it becomes stuck in the local maximum.

In our implementation, we would first complete the board by choosing moves for the blank spaces that would satisfy the constraints. If no move was available, a random number would be assigned. By allowing a large portion of the cells to be filled by a constraint-satisfying move, scores became much lower when they were all random. After each cell was filled, a random cell would be chosen that was previously blank, and a new move would be done to try to improve the score. If after 4000 iterations the score did not improve, a reset would be done to the original game. This was in the hope of a better trace being completed that would solve the puzzle. Improvements in hill-climbing would have been to develop methods that would choose the least constraining cells and values.

Algorithm Comparisons

This project includes three of our own sudoku-solving search algorithms, with the background of a particularly fast solution provided by Professor Hunsberger. In the beginning, it

was obvious none of our implementations were going to outperform the solutions, however, it was about understanding the significance of various algorithms and optimizations. During the implementation of Backtracking, Simulated Annealing, and Hill Climbing there had become apparent connections between the three. Obviously they still each have their own spin on things, like random restarts in HC or probability functions in SA. However, there were a number of key underlying techniques as well as issues with each algorithm that proved why optimizations seen in the ultra-fast sudoku solvers are necessary.

For example, in both hill climbing and simulated annealing we found that the basic setup of the algorithms would often struggle towards the end of the solving process. This is due to some of the randomness that is included within each algorithm, so it does not allow the program to target the specific cells or constraints that need to be fixed at the end of each puzzle. Hill Climbing experiences similar challenges, particularly as the game state space increases. This was an important discussion within the group, for each it took time to understand exactly where each algorithm would begin to struggle and why. These discussions lead to the changes that became the implementations that were described above for each search algorithm.

Testing & Results

Sudoku solvers can have a wide range of testing strategies based on the size of the puzzle and its difficulty. Our goal is to demonstrate the ability for each of the algorithms to solve puzzles of varying complexity. Backtracking running in $O(9^m)$ causes its testing graph to be exponential. It is very effective when running on a small number of unfilled boxes, and has very poor performance as this increases. In our tests, inputs that included a large number of unfilled boxes took an unreasonable amount of time to complete successfully, and a small number of unfilled boxes were solved very quickly. Compared to the given solutions, normal backtracking

does not work very efficiently without optimizations. Future work on the project would include different optimizations to test them against each other.

Hill-climbing testing would see improvements in scores but rarely find solutions for more difficult larger problems. Smaller easier problems could be solved using the algorithm, but many large boards such as the example tests provided in the original implementation would not get solved. This could be improved by better methods of choosing values for the cells, as a random method was used if there wasn't a constraint-satisfying move. It also would have been improved by a better method of picking the next cell to fill, as this was also random. There were a number of similar optimization techniques that could have been applied to the algorithms we implemented, we deliberated closely on which optimizations to include and what was unfeasible for the scope of this project.

We tested the *sim-annealing* on each of the test puzzles (*t1* - *t8*) by using the *test-sim-annealing* function. It takes in a puzzle, a number times to test it, and the number of steps per-test. While our algorithm didn’t find a solution every time for every puzzle, once we tuned the temperature and step length it began finding solutions sem-frequently. The lowest number of steps for a solution was 97 and most were reached after 2 or 3 hundred steps. Improvements in *sim-annealing* could have come from more careful implementation according to runtime efficiency. Our cost-func could have possibly been modified to only count the changes in a new-state, however, that was beyond the scope of the project. Additionally, further investigation into the relationship between temperature, cooling, and search length could reap fruitful results for the search. Specifically, we toyed around with selecting only constrained indexes to generate in the neighbor function when the search got stuck, however, this felt like an

additional heuristic beyond simple simulated annealing, so we decided to leave for another project.

Conclusion

This project demonstrates the incredible variety and possible applications of artificially intelligent search algorithms. Starting with the implementation of backtracking it fostered ideas on how to optimize the search to reach solutions in less time. The Professor's solutions utilize many of these techniques within the context of forward checking to provide one of the fastest AI sudoku-solving algorithms possible. Backtracking was quite the opposite, it served well when the search space was small but it exponentially increased as the puzzle became more difficult. Therefore, when implementing simulated annealing and hill climbing there was already precedent for the "typical" algorithm so that we could approach these two with the focus on improving their search from backtracking. As mentioned before, each algorithm has its own methodologies for conducting the search, but we found common struggles across the algorithms. For example, we often found ourselves having a discussion about consistency and targeted searches, which had been implemented in Professor's solutions. Therefore, we did not want to add in too many optimizations and turn the algorithms into forward checking, but it was interesting to see how each optimization could fit into the process of the algorithms. Overall, the group was exposed to a number of Constraint Satisfaction Problem-solving methodologies that showed the similarities between the algorithms, and how each could implement optimizations to improve solutions.

Works Cited & References

Chi, Eric C., and Kenneth Lange. "Techniques for solving sudoku puzzles." *arXiv preprint arXiv:1203.2295* (2012)

Davis, Ernest, et al. Artificial Intelligence: A Modern Approach. United Kingdom, Pearson, 2010.

Job, Dhanya, and Varghese Paul. "Recursive backtracking for solving 9* 9 Sudoku puzzle." *Bonfring International Journal of Data Mining* 6.1 (2016): 07-09.

N. A. Hasanah, L. Atikah, D. Herumurti and A. A. Yunanto, "A Comparative Study: Ant Colony Optimization Algorithm and Backtracking Algorithm for Sudoku Game," *2020 International Seminar on Application for Technology of Information and Communication (iSemantic)*, Semarang, Indonesia, 2020, pp. 548-553, doi: 10.1109/iSemantic50169.2020.9234267.

T. K. Moon and J. H. Gunther, "Multiple Constraint Satisfaction by Belief Propagation: An Example Using Sudoku," *2006 IEEE Mountain Workshop on Adaptive and Learning Systems*, Logan, UT, USA, 2006, pp. 122-126, doi: 10.1109/SMCALS.2006.250702.