Comparing BFS, Genetic Algorithms, and the Arc-Constancy 3 Algorithm to solve N Queens and Cross Math

Peter Irvine
College of Science And Engineering
University of Minnesota
Minneapolis, Minnesota 55455
Email: irvin124@umn.edu

***Abstract*—We compared Breadth First Search, Genetic Algorithms, and AC3 Algorithms with the test cases of N Queens and Cross Math. To do this, we created a bash script that ran a large amount of experiments with different constraints on the problems and logged CPU usage, time to completion, and memory consumption of the program and stored the results into a text file to be read later. We found that our genetic algorithm was not quite optimized and was not that useful as BFS ran in a similar if not better time. For future work, we plan to make the algorithms more efficient and rerun most of our tests.**

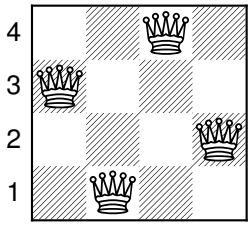
I. Introduction

Puzzles have always been a great source of entertainment for people, whether it has been casual or competitive play or just a good time waster while trying not to do homework. There are many different kinds of puzzles in existence today but in this paper we will be focusing on just two: N Queens and Cross Math.

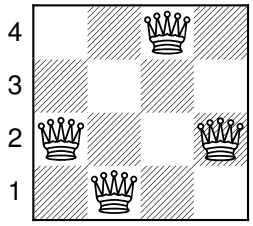
II. Background

A. Puzzles

1) N Queens: N Queens is a game played on a chessboard but the only pieces that are used are Queens. The goal of the game is to place as many queens as there are rows. However, the rules of chess still apply, meaning that you can not place a queen where she will be captured by another queen on the board. Both a valid board and an invalid 4x4 board can be seen below:



A Valid N Queens Board (4x4)



An Invalid N Queens Board (4x4)

2) Cross Math: Cross Math is a game like Sudoku, however the board is set up a bit differently. Instead of just placing numbers on a grid and making sure they don't repeat in a column, row, or box the player has to solve equations that appear on the board. In Cross Math, you do there are no repeated numbers, so for a 3x3 board, the numbers one through nine are only once. The goal of the game is to solve the equations in the rows and columns. The boards are laid out so that in between the blank boxes, there are operators that must be used to get the number after the equals sign at the end of the row or column.

In Cross Math, the rules of operations (PEMDAS - Parenthesis, Exponents, Multiplication, Division, Add, Subtract) do not apply. Instead, it proceeds to go left to right or top to bottom. This greatly simplifies the logic that is needed to solve these puzzles. Cross Math puzzles can range in size from a 2x2 board to as large as you want to make them, but for our purposes, we only tested up to a 6x6 board.

	+		+		=	15
+		×		÷		
	+		×		=	24
−		−		÷		
	+		−		=	14
=		=		=		
3		12		4		

An Unsolved Cross Math Board

4	+	3	+	8	=	15
+		×		÷		
5	+	7	×	2	=	24
−		−		÷		
6	+	9	−	1	=	14
=		=		=		
3		12		4		

A Solved Cross Math Board

B. Algorithms

1) Breadth First Search (BFS): Breadth First Search (BFS) is a very simple search algorithm for graphs. It works by "systematically explores the edges of G to discover every vertex that is reachable from s. It computes the distance (smallest number of edges) from s to each reachable vertex. It also produces a breadth-first tree with root s that contains all reachable vertices. For any vertex reachable from s, the simple path in the breadth-first tree from s to corresponds to a shortest path from s to in G, that is, a path containing the smallest number of edges. The algorithm works on both directed and undirected graphs"[4]. It is also called BFS because instead of fully exploring branches like a depth first search, it only explores one "level" at a time.

This means that when BFS finds a node, it finds the most efficient path to that node, because it has visited all other nodes up to that level and hasn't found the desired node. Below is the pseudo code for BFS[4]:

function BREADTH-FIRST-SEARCH(*problem*) **returns** a solution, or failure

```
node ← a node with STATE = problem.INITIAL-STATE, PATH-COST = 0
if problem.GOAL-TEST(node.STATE) then return SOLUTION(node)
frontier ← a FIFO queue with node as the only element
explored ← an empty set
loop do
    if EMPTY?(frontier) then return failure
    node ← POP(frontier)   /* chooses the shallowest node in frontier */
    add node.STATE to explored
    for each action in problem.ACTIONS(node.STATE) do
        child ← CHILD-NODE(problem, node, action)
        if child.STATE is not in explored or frontier then
            if problem.GOAL-TEST(child.STATE) then return SOLUTION(child)
            frontier ← INSERT(child, frontier)
```

We used BFS as a benchmark for our tests, this a brute-force attempt to solve the problem. In theory, this method should take the longest and use the greatest amount of memory and CPU.

2) Genetic Algorithms: A Genetic Algorithm is an algorithm that makes use of a heuristic to reduce the population of your answer. It does this by mutating the "organisms" or solutions of the problem. To do this it compares where it is in relation to where it needs to be, and adapts to try to best mirror the goal state. The mutations to the population are done in what's called a heuristic algorithm. The algorithm should terminate when it equals the goal state, which for this purpose, is when the puzzle is solved. Below is the pseudocode for our genetic algorithm:

function GENETIC-ALGORITHM(*population*, FITNESS-FN) **returns** an individual

```
inputs: population, a set of individuals
        FITNESS-FN, a function that measures the fitness of an individual

repeat
    new_population ← empty set
    for i = 1 to SIZE(population) do
        x ← RANDOM-SELECTION(population, FITNESS-FN)
        y ← RANDOM-SELECTION(population, FITNESS-FN)
        child ← REPRODUCE(x, y)
        if (small random probability) then child ← MUTATE(child)
        add child to new_population
    population ← new_population
until some individual is fit enough, or enough time has elapsed
return the best individual in population, according to FITNESS-FN
```

function REPRODUCE(x, y) **returns** an individual

```
inputs: x, y, parent individuals

n ← LENGTH(x); c ← random number from 1 to n
return APPEND(SUBSTRING(x, 1, c), SUBSTRING(y, c + 1, n))
```

[4]

We tested the Genetic Algorithm because we wanted to investigate an algorithm that was able to adapt to the current state of the puzzle while hopefully reducing the possibilities the algorithm must go through. In theory, this algorithm should run faster than BFS because it actively reducing the amount of options the algorithm has to go through.

3) Arc-Consistency 3 (AC3): Arc-Consistency 3 or AC3 is a constraint satisfaction algorithm; it works by satisfying a myriad of constraints that are passed into it. It operates with Variables, Domains, and Constraints. A variable can take any of several discrete values; the set of values for a particular variable is known as its domain. A constraint is a relation

that limits or constrains the values a variable may have. The constraint may involve the values of other variables.

AC3 works great for small problems that don't have many constraints and only one solution. This is why we decided not to implement AC3 for N Queens. There are many different solutions for a given N Queens problem and AC3 would not be able to return a solved puzzle. As it is, AC3 was not able to return a completely solved Cross Math puzzle - only parts of the puzzle are able to be solved the rest of the squares have a greatly reduced set of numbers that can go there.

Below is the pseudocode for an AC3 Algorithm

function AC-3(csp) **returns** false if an inconsistency is found and true otherwise
  inputs: csp, a binary CSP with components (X, D, C)
  local variables: $queue$, a queue of arcs, initially all the arcs in csp

  while $queue$ is not empty **do**
    (X_i, X_j) ← REMOVE-FIRST($queue$)
    if REVISE(csp, X_i, X_j) **then**
      if size of $D_i = 0$ **then return** $false$
      for each X_k **in** X_i.NEIGHBORS - $\{X_j\}$ **do**
        add (X_k, X_i) to $queue$
  return $true$

---

function REVISE(csp, X_i, X_j) **returns** true iff we revise the domain of X_i
  $revised \leftarrow false$
  for each x **in** D_i **do**
    if no value y in D_j allows (x,y) to satisfy the constraint between X_i and X_j **then**
      delete x from D_i
      $revised \leftarrow true$
  return $revised$

4) Previous Research: Genetic algorithms have been used for many things, from optimization to learning rules to simulating life forms. It has quite the application base, and one of those applications is solving puzzles. Ayad M. Turky and Mohd Sharifuddin Ahmad from the University of Anbar, Iraq used a genetic algorithm to solve the N Queens problem [1]. What they found was that they could solve a 2000x2000 N Queens board (on average) in about 1023 seconds. They concluded that N Queens is a problem that cannot be reasonably solved by a deterministic model and need some sort of heuristic to solve properly [1].

Rok Sosile and Jun Gu[2] also decided to look at the N Queens problem. They however were looking at the total number of moves it would take to solve any given problem. They broke it into steps, the first step was placing the first queen, which can be represented by:

$$f(x) = \int_0^x \frac{1}{1-p(x)}dx \tag{1}$$

Where f(x) is the number of steps to placing the first queen and p(x) is the probability that a random queen on column z is attacked by some previously placed queen to the left. The probability ($p_1(x)$) that a random column x is attacked by a previously placed queen can be calculated by the following equation:

$$p_1(z) = 2 * \int_0^x y + x - yxdy + \int_x^{1-x} 2x - 2x^2 dy \tag{2}$$

Which equals

$$2x - 2x^2 + x^3 \tag{3}$$

Finally, the number of steps needed to place the final queen can be expressed by the equation below:

$$P = 8 * \int_0^{0.5} \int_0^x 1 - x + x^2 + y - y^2 dydx \tag{4}$$

They used Efficient Local Search and they found the average run time for a 10^4 board to be 0.1 seconds, for a 10^5 board to be 1.1 seconds, for a $2 * 10^6$ board to be 17.0 seconds, and for a $3 * 10^6$ board to be 54.7 seconds [2]. This proves that the N Queens problem can be solved with a very large board in a reasonable amount of time.

In "A New Solution for N-Queens Problem using Blind Approaches: DFS and BFS Algorithms," the authors investigate the N Queens problem with BFS and DFS searches[3]. Their results are a bit slower than ours are. They solved a 9 queen board in 276 seconds, while we solved it in 11.09 seconds [3]. The big difference lies in the 10x10 board. It took them 7149 seconds to solve the puzzle, while it only took us 54.06 seconds. However, they did show that DFS is much faster than BFS which makes sense. With a DFS search branches are explored until the solution is found, whereas with BFS, the levels are gone down one at a time, and the number of nodes to keep track of grows almost exponentially. BFS also takes up more memory as it has to remember more nodes than DFS does.

III. Experiment Methods

In order to see which algorithm worked best for each puzzle we ran a series of tests on both algorithms with both problems. To test this, we started with a small board and increased the size until the program could no longer run in a reasonable amount of time or was killed by the system for using too much memory. The following data was collected: how much memory the program was using, the amount of time it took for the program to run, and the amount of the CPU allocation we were given the process was using. To actually run the experiments and log the results, we wrote a bash script that ran the python calls and stored the output of the program and results of the test to a file that was named according to which algorithm and puzzle was being solved. For example, the name of a Binary search on an N Queens board of size 5 is:

```
output_bn05.txt
```

A sample of that script can be seen below:

```
{ date ;} >> output_bn05.txt
{ /usr/bin/time python3 pt2solver.py bfs\
    nqueens 5 ; } 2>> output_bn05.txt
{ printf "\n" ;} >> output_bn05.txt
    //The slash after bfs is indicating a
    //new line that has to be put into the
    //paper that is not in the actual code.
    //The code snippet is too long to fit on
    //in the paragraph.
```

A sample output of the experiments can be seen below:

```
Mon Oct 24 14:29:00 CDT 2016
```

```
0.06user 0.00system 0:00.08elapsed 92%CPU\
        (0avgtext+0avgdata 11396maxresident)k
0inputs+0outputs (0major+1341minor)pagefaults\
        0swaps
```

The output starts by printing the date and time the test was started. For the experiments that were not completed, we just have the date and time the experiment was started. The next line shows the time lapsed (in hh:mm:ss.ss), the peak CPU usage, and the amount of memory usage. The memory usage statistic used was the maxresident option which is given in Kilobytes (k).

To test each algorithm we needed to create different boards for the programs to be run on. For N Queens that was very straight forward, only the board width and height and the algorithm will solve the puzzle from there. For Cross Math, different board sizes and different board puzzles must be created. To determine the actual puzzles that were on the board we ran a script that was given to us by the professor (Dr. Amy Larson). This script takes in board size and then randomly makes a solvable board for the algorithm to solve.

We expected our algorithms to take some time. Both BFS and Genetic can take a long time to complete certain tasks. However, we originally thought that the Genetic Algorithm would solve both puzzles faster than BFS would, but in this case it did not. BFS is expected to run in $o(n^2)$ time and the Genetic Algorithm to run in $O(O(Fitness)*(O(mutation)+O(crossover)))$ time.

The experiments were coded in Python3 and were running on the College of Science and Engineering's Vole and Atlas servers at the University of Minnesota. These are both Linux based servers. Vole is running on a Virtual 8-core Intel Xeon CPU E5-2695 running at 2.30GHz with 36GB of ram and two NVIDIA Grid K2 graphics cards. Atlas is running on an AMD Opteron CPU 6272 running at 2.10GHz with 256GB of ram.

IV. Experiment Results and Analysis

After running our script, we put the results into tables a graphs that can be seen below.

Table for Cross Math:

Cross Math			
Size	CPU Usage(%)	Time	Memory
BFS 1x1	48	0.12	11232
BFS 2x2	43	0.14	11084
BFS 3x3	62	0.08	11236
BFS 4x4	88	60	86480
~~BFS 5x5~~	~~65~~	~~14:01:15~~	~~9502868~~
Genetic 1x1	64	0.05	11176
Genetic 2x2	80	0.06	11156
Genetic 3x3	72	0.1	11224
Genetic 4x4	89	2568	269348
~~Genetic 5x5~~	~~65~~	~~14:01:01~~	~~380124~~
AC3 2x2	82	0.05	10832
AC3 3x3	87	0.06	12720
AC3 4x4	96	0.48	12624
AC3 5x5	99	148	12712

Table for N Queens:

N Queens			
Size	CPU Usage(%)	Time	Memory
BFS 4x4	40	0.12	11192
BFS 5x5	61	0.11	11228
BFS 6x6	76	0.18	11284
BFS 7x7	76	0.57	11300
BFS 8x8	83	2.19	11640
BFS 9x9	83	11.09	13212
BFS 10x10	91	54.06	21540
Genetic 4x4	44	0.12	11240
Genetic 5x5	77	0.1	11456
Genetic 6x6	65	0.08	11328
Genetic 7x7	59	0.55	11692
Genetic 8x8	87	23.47	66068
Genetic 9x9	91	22.34	66716
Genetic 10x10	87	1584.7	3642732

In the Cross Math table, BFS 5x5 and Genetic 5x5 are crossed off. That is because we let the program run overnight and the system shut it down. The last data point that we got was around 14 hours for both algorithms. We believe the system shut us down because we used up too much memory and ran out of our allotment of memory. Below there are graphs comparing runtime, CPU usage and Memory - providing a graphical comparison of the data.

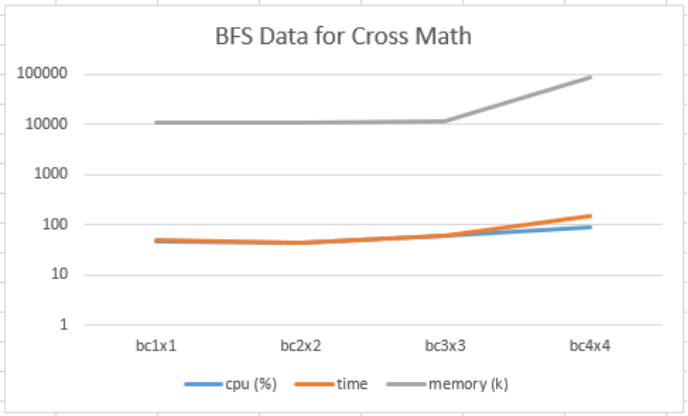


Fig. 1. Graph of BFS Data for Cross Math

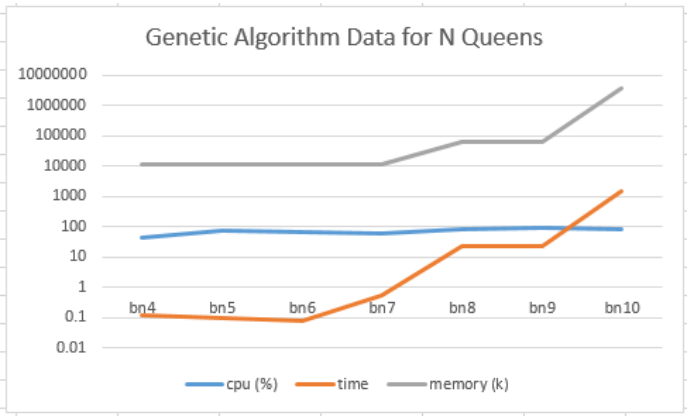


Fig. 2. Graph of BFS Data for Cross Math

As seen in the data and graphs above (and some below), BFS has the slight advantage in both of the puzzles (Cross Math and N Queens). This is not it what is supposed to happen. The reason that genetic algorithms exist is so they can mutate based on the current population and make the pool of options smaller. In our case, it seemed as if at some points the population of our algorithm actually got bigger.

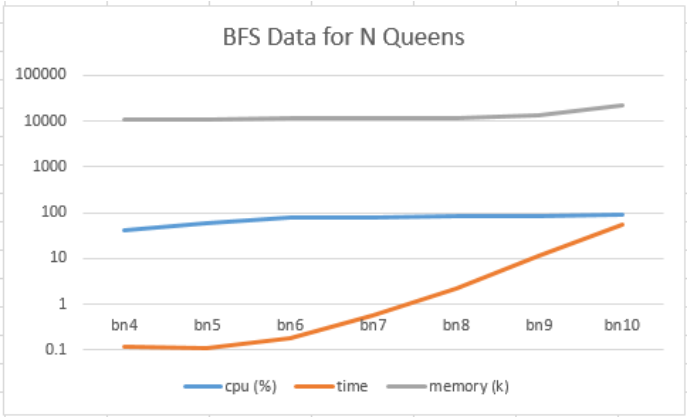


Fig. 3. Graph of BFS Data for Cross Math

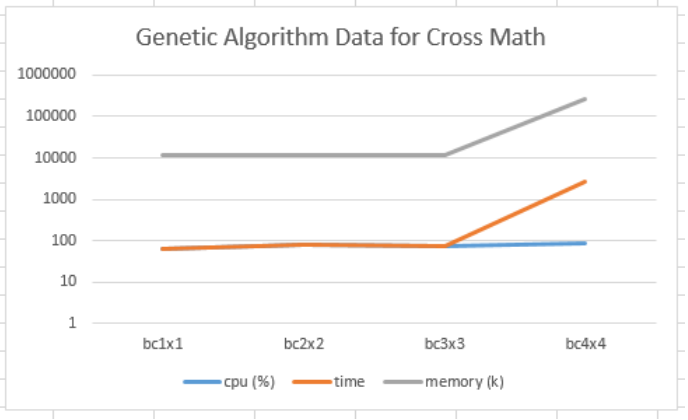


Fig. 4. Graph of BFS Data for Cross Math

We could have fixed this by setting a max population size. This might contribute to why our genetic times were so much larger when the board sizes got larger - the populations grew almost exponentially. In theory, the genetic algorithm - on average will run much faster than a BFS search, because the BFS doesn't have a heuristic to prune unwanted results. What this shows is that our heuristic isn't pruning enough.

BFS does have one advantage over Genetic Algorithms: it is consistent. When run multiple times over the same puzzle, BFS should return in the same amount of time every time. With a Genetic Algorithm it could vary each time, based on how the program mutates. If it mutates in a way that is favorable to the program, it will decrease the run time of the algorithm. If it doesn't mutate in a favorable way, it will increase the run time.

AC3's results were not the same as the ones above. While we did measure the run time, CPU usage, and Memory usage we also measured success rate, which can be seen in the table below:

AC3 Success Rate on Cross Math		
Board Size	Successes	Failures
2x2	5	45
3x3	36	14
4x4	22	18
5x5	0	50

What this data shows is the amount of times the algorithm completely solved a problem. It would only "solve" a problem if there was only one solution to the puzzle. Because it is hard and rare for puzzles to have only one answer, AC3 fails quite often - as seen in the table above. It seemed to be quite effective for our 3x3 and 4x4 tests, but for 2x2 and 5x5 tests it did quite poorly. For the 2x2 it failed most of the time because it is quite difficult to make a puzzle with only one solution and have it still be a valid puzzle. For the 5x5 puzzles there are so many different combinations that it is almost impossible for AC3 to solve them (which is seen in the data above). Because

of this, we decided not to test a 6x6 Cross Math puzzle; it would have been too large for AC3 to solve reliably. It also would have taken much longer to run as there are quite a bit more possibilities for solved puzzles.

What this shows is that AC3 is not a very efficient search algorithm for problems that have multiple solutions. For problems that have a single solution (generally) like Sudoku, it would be perfect because of its speed. For puzzles that have multiple solutions, the AC3 algorithm would be great as a preprocessor for other search algorithms that could actually solve the problem. AC3 would reduce the population size immensely allowing another algorithm to run much quicker than you could before. If the remaining population was put into BFS, it would greatly reduce the amount of time it would take for BFS to finish.

V. Conclusion and Future Work

After running our experiments we found that our results for the Genetic Algorithm did not line up with the theory. We think that this is because of how we programmed the algorithm and heuristic. After analysis, we discovered that our population size was increasing instead of decreasing. This was due to an inefficiently written heuristic and solving algorithms.

BFS on the other hand worked the way we expected it to. It was able to complete its searches in a reasonable amount of time and didn't rely on a heuristic to try to make it run faster (or slower if the heuristic isn't done properly). This served as our benchmark for comparing the other types of algorithms because this doesn't have anything to modify it's population. This is essentially a brute force method of solving the puzzle.

Arc Consistency-3 worked as we expected it to for Cross Math. We were not able to put N Queens through AC3 because it does not have a single solution. Problems that don't have a single solution are unsolvable through AC3. While we were only able to get AC3 to fully solve smaller puzzles, it is a great way to quickly reduce the population size to put into the more extensive search algorithms.

Future work for this project will be to make the algorithm and heuristic for the Genetic Algorithm better and more efficient. We also plan to do more trials over a larger data set (ie, larger boards), and plan to run those tests multiple times to get an average for all of our stats. Another avenue we could take would be combining AC3 with another search algorithm. The purpose of this would be to see if reducing the initial population size would reduce the run time of the overall search algorithm.

Acknowledgments

The author would like to thank Alex Oelke, Lane Scherber, and Ryan Reding who were in the group from which the data was collected. A special thanks also goes to Elise Lohmann for proofing the paper and fixing the many grammatical errors that this paper had.

References

[1] Ayad M.Turkyl and Mohd Sharifuddin Ahmad "Using Genetic Algorithm for Solving N -Queens Problem.", University of Anbar,Iraq.
[2] Rok SosiE, Member, IEEE, and Jun Gu, Senior Member, IEEE. "Efficient Local Search with Conflict Minimization: A Case Study of the n-Queens Problem"
[3] Farhad Soleimanian Gharehchopogh, Bahareh Seyyedi, and Golriz Feyzipour. "A New Solution for N-Queens Problem using Blind Approaches: DFS and BFS Algorithms"
[4] Introduction To Algorithms