# Search Algorithms

Nathan Wuiske

## Problem Statement

This assignment is based on solving the n-queens problem using the Hill-climbing search, Random re-start hill climbing search, Simulated annealing search and Genetic algorithm search. The following algorithms are implementing in the C++ programming language using “Codeblocks” – a powerful IDE tool to test and write C++.

## Core code explained

Function Solve  
Within the program is code that is reused for every algorithm implemented in this assignment. The first function that is used is called *solve* that takes an argument *gameSize* which is the size of the chess board. Inside this function the main data structure to contain the chest game positions is created:  
 *int \*game;*

*game = new int[gameSize];*

The first expression is used to allocate memory to contain one single element of type int. The second expression is used to allocate an array of elements of type int, where *gameSize* is an integer value representing the amount of space in the array. It is unessessary to use a 2D array to store the game as it would be wasted space, because the chessboard is always going to be *nxn* and the number of queens is equal to the size of the board. A random state is generated using the *randomGameState* function so that the board has values. This is then following by a while loop which checks if the heuristic score of the Game is not equal to 0, therefore this loop will execute until a good solution is found. Inside the while loop is an if statement checking if the return value from the *stateNext* function is false. If *stateNext* returns false then it will reset the board state by calling *randomGameState* and try searching for a solution again. When stateNext returns true, it means that a solution has been found which terminates the while loop and continues on to print out the result. (Used for hill climbing searches)

Function randomGameState

The randomGameState function takes arguments *game* and *gameSize*. This simple function generates an initial random state of the board by randomly placing a queen on each row. This is done by using a forloop that generates a random number between 0-*gameSize* and placing that within *Game*[i].

Function checkAttack

The checkAttack function takes arguments *game* and *gameSize*. This function calculates the heuristic score of the board and is used many times later on to compare better board states. Inside the nested forloop is a check statement to determine if a queen lies on the same column or diagonal. It doesn’t test for horizontal because by nature of the data structure there can’t be anything horizontal, each row can only hold one queen. To find diagonals *row-col* and *row+col* is used. For example, if a queen was on (1,2) using the formula *row-col* it would then be 1-2=-1, therefore any other square where *row-col* is equal to -1 will be diagonal to this queen such as (2,3) = 2-3=-1. This works similarly with finding the other diagonal using *row+col.*

Function stateNext

The *stateNext* function takes arguments *game* and *gameSize*. This function compares the heuristic score from the Hillclimbing algorithm and it compares it with the current heuristic score (*Game*). If the hillclimbing heuritic score is better (lower) than *Game*, *Game* gets the queen locations of the hillclimb algorithm. If this is not the case, then this function returns false which is checked in an if statement in the *solve* function. This is why this function returns a *bool*.

## Algorithm Explanations

Random Restart Hill climbing Search  
This algorithm works by checking every possible move and returning the best of these. The heuristic cost can be calculated after one move on the board. The move with the lowest heuristic cost is selected and this process is repeated, therefore it will be “climbing a hill” until the peak which is *h=0.* The hill climbing takes in arguments *game* and *gameSize*. To begin with, a vector was created.   
*vector<int> container;*  
The reason why a vector was used instead of an array is because vectors can change in size and are handled automatically by the container. There are also functions that can be used on vectors(such as clear, push, size) that can be easily used in comparison to an array. This vector will be used for holding the best game result. A new dynamic array was created as well called *gameOut*:  
*int \*gameOut;*

*gameOut = new int [gameSize];*This will be used to compare against *Game* later on to see which holds a better heuristic cost. The contents of *gameOut* are set equal to the contents of *Game*. The core part of the algorithm contains a nested forloop. Within the outer loop the vector *container* is cleared so that no garbage is in it and the *gameOut* contents are “pushed” into it.   
*container.clear();*

*container.push\_back(gameOut[i]);*  
It doesn’t matter if *Game* or *gameOut* are passed into the vector, as long as the vector has the game contents the algorithm will work as planned. Within the inner loop the index of *gameOut* is set to *j* so *gameOut* now has a different set in comparison to *Game*. A variable called *newhCost* is created which uses the *checkAttack* function with the input paramenters being *gameOut* and *gameSize*.  
*newhCost = checkAttack(gameOut, gameSize);*At the beginning the *htoBeat* variable was also created, but instead inputs *Game* instead:  
*htoBeat = checkAttack(game, gameSize);*The next part of the algorithm tests these two cases by using comparisons. If they both are equal then vector container will simply get *j*, because it doesn’t matter what *container* is filled with as both sides are equal. If *gameOut’s* heurisitc cost is less than *Game* then it’s a better move, therefore we clear the container, add *j* and set *gameOut* equal to *Game*. There doesn’t need to be another comparison check, if *GameOut* heuristic cost is greater than *Game* then do nothing. Now a random best move is selected using the *rand*() function, which will set *gameOut* equal to the vector *container* and the vector’s index will select randomly between 0-*container.size* (the size of the vector).   
*gameOut[i] = container[rand() % container.size()];*  
After this the best solution is found and *gameOut* is returned and passed into the *stateNext function* (see stateNext description). This is the part where is “random restart” takes place.

Hill climbing Search

The hill climbing algorithm is similarl to the random restart algorithm, however with random restart it can become somewhat expensive to check every possible square for the best moves. Instead of finding the best move at every step, hill climbing will simply find a better move.

This is a much more simplified version. The main changes is that now we do not need a container to contain the best moves and most of the parts from random restart can be removed. The algorithm contains one check statement, that is, it checks if the newhCost (gameOut) is less than htoBeat (Game), if so it will return the first better match found.

Simulated Annealing Search

Simulated Annealing search is meta-heuristic, which means that it doesn’t guarantee a solution since it searches for a good approximation in a small amount of time by iteratively improving it. The algorithm starts with a given temperature and a random solution and iteratively calculates a new random solution. A solution is selected depending on the result of the probability function that takes into account if the solution is better or worse than the current accepted solution and the temperature. As the temperature decreases, the probability of accepting worse solutions also decreases. The temperature is basically an indicator of how long it can run. Simulated annealing basically "heats up" then "cools down". In the beginning, "bad moves" will be taken based on a high temperature. As the temperature goes down, bad moves should not be taken as much. When the temperature is 0, the algorithm should come to a stop and output what it has. The probability function used to decide if a move with a negative impact on the score is used within this algorithm:  
 *P(Ed)=exp(-Ed/T)*

where *Ed* is the “energy” difference (the difference in the number of attacked queens) and *T* is the temperature for the round. However this algorithm will take a modified apparoach using:  
*P(Ed)=exp(-Ed\*T)*Within the forloop there will be an expression *t=t\*decayRate* which will slowly decrease *T* over time, much like using the first function.

The initial variables *t*, *decayRate*, *energyDiff, tempGame[gameSize]* and *ran* (randomly generated number) are created for this problem.

First off, the *tempGame* is set equal to the contents of *Game*, then a random move is selected from *tempGame* using the *rand* function.  
*tempGame[rand()%gameSize] = rand()%gameSize;*  
This is to gain a random board state. Next, the energyDifference is calculated using:

*energyDiff = checkAttack(tempGame, gameSize) - checkAttack(game, gameSize);*Which simply gets the heurisitic cost difference between *tempGame* and *Game* and will be the “energy” within this problem.   
Next an if statement will check if the random integer (ran) is less than *P(eD)=exp(eD\*T),* if this is true then it will take the bad move by setting *Game* equal to *tempGame*.  
Another if statement checks if the *Game* heuristic score is equal to 0, if so then a solution has been found and the results are printed out.   
Something that is important that needs to be noted is that it is possible for this algorithm to get “stuck”, which is also known as a plateau. Therefore, it is usually a good idea to put a cap on the number of attempts the algorithm can make before giving up. In the annealing algorithm, the number of iterations was set to 200,000 iterations.

Genetic Algorithm Search  
For the genetic algorithm to work properly there are key variables that need to be implemented. Firstly the initialization, which contains the randomly distributed population that is generated. The nature of the algorithm is that lower population size will lead to lots of time in computation of approximate solution whilst a higher value will cause internal iterations to increase. A 2D array was created which will be the population:  
*int pop[gameSize][gameSize];*  
Where the first parameter is population size and second parameter is size of the chess board. Similarly this was done for the child which will accessed later on:   
*int child[gameSize][gameSize];*

The *pop* array is then populated randomly using the rand() function, where the random number will generate between 0-*gameSize*.

The rest of the code will be executed within a *whileloop* and will continuely do so until a solution has been found. Using a variable called *exitCode*, while it is equal to 0 the loop will continue to run, to break this loop an if statement will check if a solution has been found and if so it will set *exitCode* equal to 1, exiting the loop and producing a solution.   
  
The way a solution is checked is by using a *checkWeight* function which acts very similar to the *checkAttack* function, but instead the “heuristic cost” (in this case it’s the weight), will start at 28 – the number of possible clashes on an 8x8 board. It will then check for corresponding queen clashes and decrememnt the weight value.   
*if(checkWeight(pop[i], gameSize) == 28)*The way this works is that it will check if the *pop* array has any queen clashes, if it is equal to 28, then there are no queens clashing because if there were, it would return a decremented value of 28. This would then print out the answer.

Next, Categorical distribution or weighted probability distribution is used to assign a probability to each outcome using its own weight and the total weight.

If you have a set of outcomes:  
8fae7e3d9ebc497231b2251f3c633e20  
And associate a value (in this case a weight) with each outcome.

0e25045916a0487b698a2339b22d1a14

Then a total weight can be calculated

46a372a5636f85273a184cffe13fab2c

A probabilitycan be assigned to each outcome using its own weight and total weight

6d4ecf4e5ce9369b273bd56b36453f56

Although this algorithm doesn’t use these exact formulas it follows a similar proccess.

A variable called *weight* is created and is used to check the weight of the population:  
*int weight = checkWeight(pop[i], gameSize);*

A forloop will then execute *weight* amount of times setting the *weightProb* index equal to *i.*   
*weightProb[weightProbLength] = i; //weightProbLength is incremented by 1 each iteration*

This will fill the array with *member* *number* and will be used later on to randomise the index of the parents for crossover and mutation.

Parent selection is done next. This is done by simply selecting 2 random, non equal parents. Two new parents are created and are assigned to the *weightProb* array with a randomly generated index of *weightProbLength* from before. Once two parents are selected, the proceeding part is crossover which involves selecting a part of parent 1 and concatenating it with parent 2. This selective addition will create an offspring that may or may not have a solution, so essentially what it’s doing is taking two solutions ( the parents) then applying crossover between randomly selected portions of the arrays in order to produce two new (and possibly better) child solutions. Two for loops are used to iterate over the concatenating. The first loop will iterate a random number of times between 0-*gameSize* whilst the second loop will iterate *gameSize.* This is to further randomise the child factors. During the crossover, the child is set equal to the population with the index being the parent.  
*child[i][j] = pop[parent1][j];  
 child[i+1][j] = pop[parent2][j];*

The second forloop will then do something similar but instead is a step forward and the order is switch:

*child[i][j] = pop[parent2][j];*

*child[i+1][j] = pop[parent1][j];*

Mutation is the final step involved within this algorithm. A mutation operator is applied on new individuals. It randomly changes few individuals (mutation probability is usually low). Statistically, mutation is very rare so in this case an if statement for mutation is used, using a random between 0-500,000 (arbitrary number) it will compare if that result is less than or equal to the *mutationProb*\*500,000:  
*if(mutationProb\*500000>=rand() % 500000)*If this becomes true then it will move onto another check. A variable called *c* is created and set randomly between 0-2.   
*int c = rand() % 2;*  
If this is equal to 0, then start standard mutation in which the child gets a random index of the board.  
*child[i][rand() % gameSize] = rand() % gameSize;*

If this is not equal to 0, then do the same but increment *i+1*

The final forloop is just setting the *population* array equal to the *child* array and the *weightProbLength* is reset to 0 to be used in the while loop again. If this didn’t reset to 0 then it would carry over a value from the previous iteration to the next, causing errors.

## Analysis of Algorithms

|  |  |  |  |  |
| --- | --- | --- | --- | --- |
| **Algorithm** | **Game Size (Size of n)** | **Run Time** | **Parameter setting(s)** | **Solution** |
| Hill Climbing Search | N=4 | 0.001 seconds | NULL | (1,3) (3,2)(0,1(2,0) |
| Hill Climbing Random restart | N=4 | 0.001 seconds | NULL | (2,3)(0,2)(3,1)(1,0) |
| Genetic Algorithm | N=4 | 0.002 seconds | mutationProb=0.5 | (1,3) (3,2)(0,1(2,0) |
| Simulated Annealing | N=4 | 0.001 seconds | decayRate=0.99 t=100 | (2,3)(0,2)(3,1)(1,0) |
| **Second Test** |  |  |  |  |
| Hill Climbing Search | N=5 | 0.001 seconds | NULL | (0,0)(3,4)(1,3)  (4,2)(2,1) |
| Hill Climbing Random restart | N=5 | 0.002 seconds | NULL | (0,0)(3,4)(1,3)  (4,2)(2,1) |
| Genetic Algorithm | N=5 | 0.003 seconds | mutationProb=0.1 | (0,0)(2,4)(4,3)  (1,2)(3,1) |
| Simulated Annealing | N=5 | 0.002 seconds | decayRate=0.90 t=100 | (0,0)(2,4)(4,3)  (1,2)(3,1) |
| **Third Test** |  |  |  |  |
| Hill Climbing Search | N=8 | 0.002 seconds | NULL | (3,1)(5,0)(7,4)(1,5)  (6,2)(0,3)(2,7)(4,6) |
| Hill Climbing Random restart | N=8 | 0.004 seconds | NULL |  |
| Genetic Algorithm | N=8 | 0.007 seconds | mutationProb=0.5 | (7,4)(3,2)(0,7)(2,0)  (5,1)(1,3)(6,6)(4,5) |
| Simulated Annealing | N=8 | 1.122 seconds | decayRate=0.99 t=100 | Couldn’t Find a solution |
| **Fourth Test** |  |  |  |  |
| Hill Climbing Search | N=20 | 0.0047 seconds | NULL | Too large to write |
| Hill Climbing Random restart | N=20 | 0.0093 seconds | NULL | Too large to write |
| Genetic Algorithm | N=20 | 7.463 seconds | mutationProb=0.5 | Too large to write |
| Simulated Annealing | N=20 | 8.125 seconds | decayRate=0.99 t=1000 | Too large to write |
| **Final Test** |  |  |  |  |
| Hill Climbing Search | N=100 | 8.861 seconds | NULL | Too large to write |
| Hill Climbing Random restart | N=100 | 9.3463 seconds | NULL | Too large to write |
| Genetic Algorithm | N=100 | Unknown – took too long to find | mutationProb=0.5 | Couldn’t Find a solution |
| Simulated Annealing | N=100 | 654.346 seconds | decayRate=0.99 t=1000 | Couldn’t Find a solution |

Hill Climbing Trends – Random restart and Hill climb

The hill climbing search works very effectively and pairs nicely with the random restart algorithm. After running the program approxiamtely 100 times for test results, it was found that 2 out of those 100 times the random restart produced a different result. From this, it seems that there is a very low 0.02% chance of the algorithm finding a worse solution than the current best one. The run times of both these algorithms are very fast in comparison to the genetic and simulated annealing searchs. Although they do take longer than what other search algorithms would take to find solutions, they produce solutions 100% of the time. From looking at the run times, random restart searching is slightly less effective, in most the tests it was seen that it would take slightly longer for the algorithm to produce the solution. The hillclimbing search is suprisingly effective for such a small algorithm in that it only checks for the first better move, and produces the same result as random restart most of the time.

Genetic Algorithm Trends  
The genetic algorithm works really well if the input size is small. As soon as N becomes moderately large the time taken to compute becomes exponential according to the results. As N=8 it takes 0.007 seconds, when N=20 it takes 7.463 seconds. From this we can see that the run time sky rockets as N increases and that this algorithm is not good in large test cases. The population is just set to the size of the board, so this might be a limiting factor for this algorithm. The mutation probably had little effect on the performance of the algorithm, as the probability got higher, the algorithm was slightly faster at run time. The way this algorithm was implemented had one glaring inefficiency and that is the *weightDistribute* array. As the board size increases this array needs to be larger to accommodate the space calculations so if a large board state is entered the array would need to be big enough for it, it’s difficult to determine what the right size is and therefore this would cause problems later on.

Simulated Annealing Trends

Simulated Annealing can become very unstable dependant on the parameters, one change makes or breaks the algorithm. Typically the temperature(t) will be set to around 100 (t=100). After testing it was found that as N increases in size, T needs to decrease in size. An example of this was a test when N=100 and T=1000, it took over 10 minutes to run! A solution was still not found. T needs to scale with N properly. After testing with the decayRate (cooldown rate), it was found that it didn’t really impact the performance of the algorithm. The decayRate was tested between 0.80 and 0.99. As the decayRate increased, a miniscule increase in performance was found when N got larger. This algorithm in comparison to the others is by far the worst yeilding poor run times no matter what parameters are used. As the problem got larger, it seemed simulated annealing got exponentially worse in time and space constraints and most the time could not find a solution at all due to the sheer random nature of the algorithm.

Overall Analysis

From analyising all these algorithms it can be seen that the most effective to least effective algorithms are Hill climb search, Hill climb random restart search, Genetic search then Simulated annealing. All algorithms work as intended when N is low, but as soon as N increases simulated annealing cannot always find a solution fast enough within the iterations and genetic algorithm can take too long to find a solution.

To put in perspective how long it takes Genetic and annealing to run the following graph shows the run time at N=20:

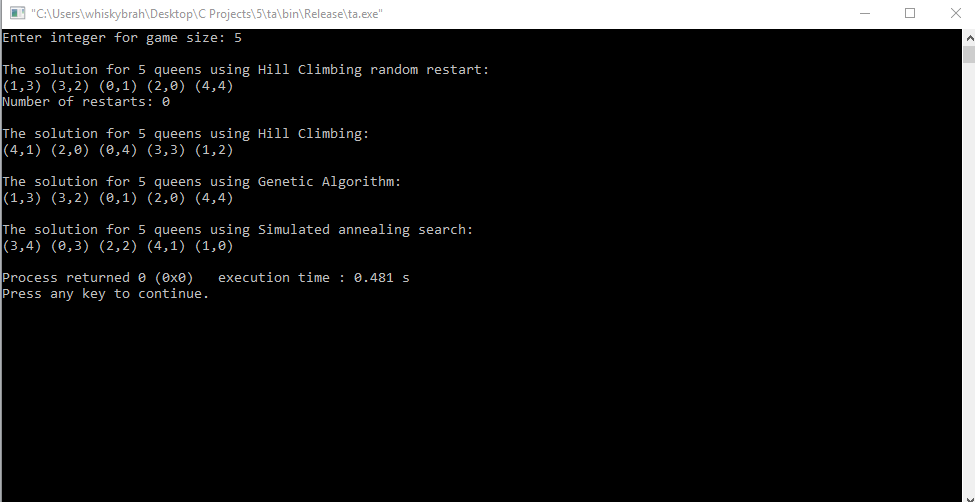
The hill climbing search run times are pretty much non-existent.

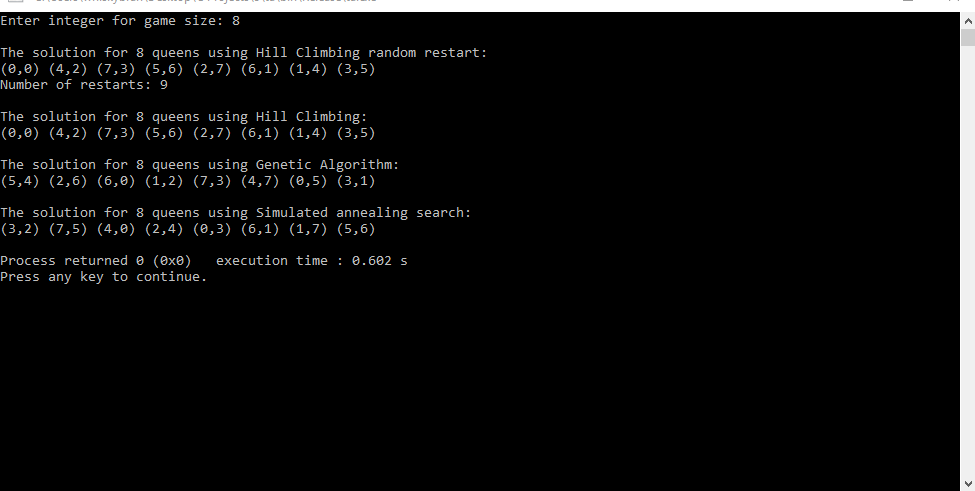
The following graph also took the run time between 5-30 queens.

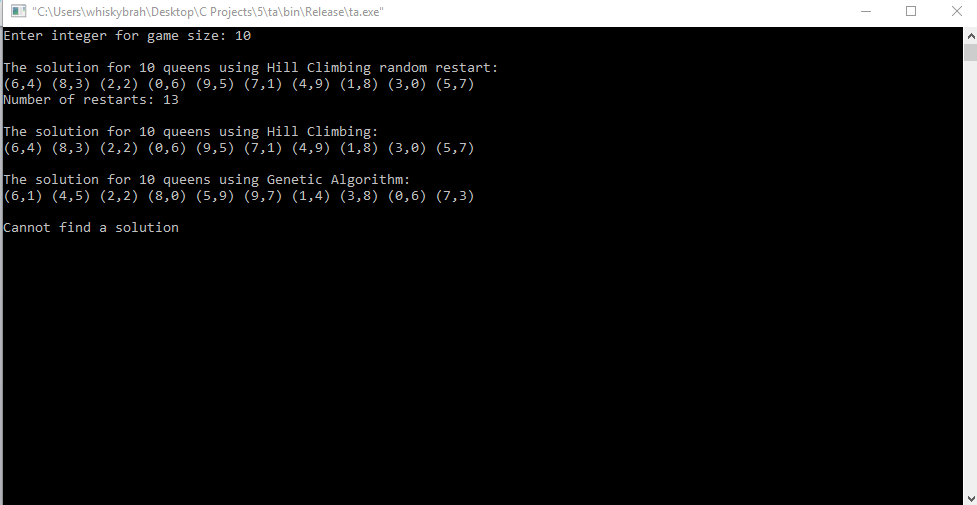
The annealing and genetic algorithms become increasingly impactful as the number of queens increases and would not be good in large test cases.

## Program Screenshots

Below are some examples of the output given by the algorithms.







Sources:

Wikipedia. 2017. Categorical distribution - Wikipedia. [ONLINE] Available at: <https://en.wikipedia.org/wiki/Categorical_distribution>

Ahmed S. Farhan. 2015. Solving N Queen Problem using Genetic Algorithm (Theory) [ONLINE] Available at: <http://research.ijcaonline.org/volume122/number12/pxc3905005.pdf>