CS 534 - Assignment 1

Heavy Queens Problem

Group members: Rutwik Bonde
Lorena Maria Genua
Tanish Mishra
Lalith Athithya Navaneetha Krishnan

1. Part 1 – Path search

The first part of the assignment will focus on solving the N-queens problem with the following search algorithms: ***Greedy Best-first search*** and ***A****.
Starting with an N by N chessboard with n queens, we need to return a solution such that none of the queens is attacking each other, vertically, horizontally or diagonally. Each queen is placed in a different column, and they can only move vertically.
Each queen is assigned a different weight from 1 to 9, hence making some of those queens more difficult to move. The cost of moving a queen is given by $w^2 b$, where w is the queen weight and b is the number of moved tiles.
(Please note that henceforth, when an algorithm "fails to solve" a board, it implies that it did not return a valid solution with 0 attacking pairs within the stipulated time of 30 seconds. This may either be because it was getting closer to a solution but timed out or that it was simply not making progress.)

1.1 Greedy Best-first search

The Best-first search algorithm uses a *heuristic function h(n)* to decide which node to explore next. This way, it tries to predict which path to follow to get closer to the solution. Specifically, to the N-queens problem, the goal is reached when the number of attacking queens (N) is 0.

1.1.1 Implementation

First, we check if the initial configuration of the board is a solution, that is if the number of attacking queens is 0. If it is not, we add the initial configuration of the board to the queue. We iterate until the queue is empty. If the queue is empty, it stops, and it returns, no solution has been found. The elements in the queue are sorted according to the heuristic function, that is the number of attacking queens for the current board configurations.
The first element of the queue, with the lowest heuristic, is removed, and it is expanded to the neighboring nodes, that is the configuration of the board by moving one queen up and down the column. For each new configuration, if it hasn't been visited already, the heuristic is computed, and the new node is added to the queue.

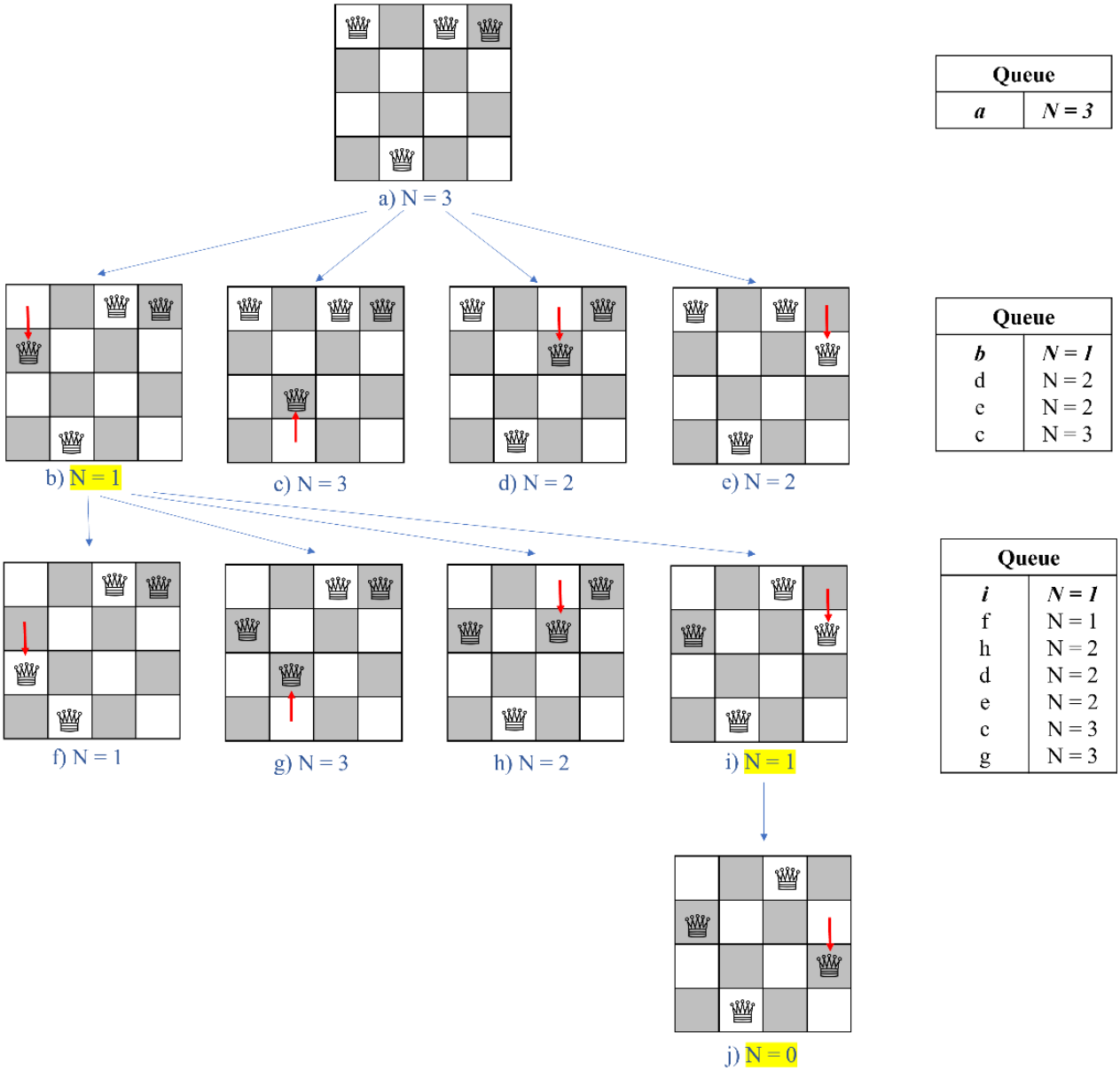


Fig. 1 – Example of 4-Queens problem with greedy search implementation

Fig. 1 shows an example of greedy search implementation for a 4-Queens problem. Starting from an initial random board configuration, the adjacent nodes are expanded, and the heuristic is computed for all of them. Next, the configuration with the lowest heuristic is chosen (b), being the first in the queue (N = 1) and the adjacent nodes are expanded, and so on. The algorithm stops when the solution j with N = 0 is found.

1.1.2 Solution

The solution for the provided 5x5 initial board was successfully returned as follows:

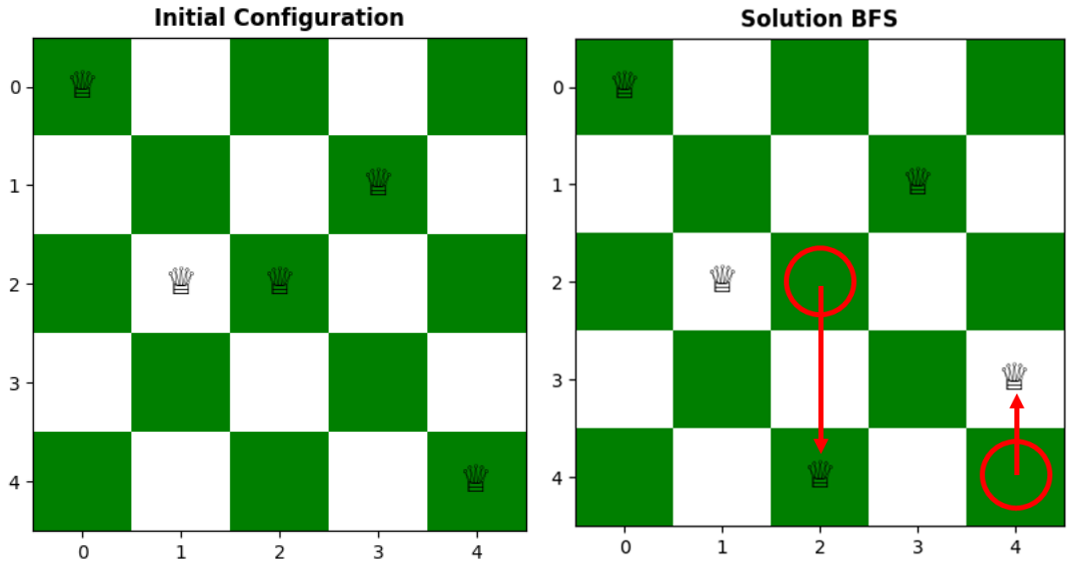


Fig. 2 – Initial board configuration and final solution using greedy search

Solution: Move column 3 down 2 squares.
Move column 5 up 1 square.
Elapsed time: 0.05 s
Nodes expanded: 165
Search depth: 4
Solution Cost: 209

1.2 A Star Algorithm

A* is a search algorithm used to find the shortest path between an initial and a final point. It was initially designed as a graph traversal problem, to help build a robot that can find its own course and it still remains a widely popular algorithm for graph traversal. A* is an optimal and complete algorithm, because it returns the least cost outcome for a problem (optimal) and returns a correct solution, if any (complete).
A* can also be implemented on weighted graphs: a weighted graph uses numbers to represent the cost of taking each path or course of action. This means that the algorithms can take the path with the least cost and find the best route in terms of distance and time.
A major drawback of the algorithm is its space and time complexity. It takes a large amount of space to store all possible paths and a high amount of time to find a solution.

So, why use A Algorithm?*
A* Search Algorithm is a simple and efficient search algorithm that can be used to find the optimal and shortest path between two nodes in a graph. It is an extension of Dijkstra's shortest path algorithm (Dijkstra's Algorithm). The extension here is that, instead of using a priority queue to store all the elements, we use heaps (binary trees) to store them. The A* Search Algorithm also uses a heuristic function that provides additional information regarding how far away from the goal node we are. This function is used in conjunction with the f-heap data structure in order to make searching more efficient.

1.2.2 Algorithm

1. Initialize the open list
2. Initialize the closed list, put the starting state on the open list (you can leave its f at zero)
3. while the open list is not empty
   a) find the state with the least f in the open list and call it "current_state"
   b) pop q off the open list
   c) generate new neighboring states (either vertically/horizontally or both together)
   d) for each new_state:
      i) if new_state is the goal, stop searching
      ii) else, compute both g and h for new_state
      new_state.g = current_state.g + distance between successor and q
      new_state.h = number of attacking queens
      new_state.f = new_state.g + new state.h
      iii) if a state with the same position as the new_state is in the OPEN list which has a lower f than the successor, skip this successor
      iv) if a state with the same position as the new_state is in the CLOSED list which has a lower f than new_state, skip this new_state otherwise, add the new_state to the open list
   end (for loop)
   e) put q on the closed list
   end while loop

1.2.3 Implementation of A Algorithm in Heavy Queens Problem*

Initially, we take the initial board state and board size as the input for the A* function. The heuristics that we have used here is the number of total attacking pairs in the current state. We treat every state as a different node. We check if the initial configuration of the board is a solution, that is if the number of attacking queens is 0. If it is not, we add the initial configuration of the board to the open list. We iterate until the open list is empty. If the open list is empty and if by then we don't find the goal state, it stops, and it returns no solution has been found. The elements in the queue are sorted according to the total f cost function, that is the number of attacking queens for the current board configurations plus the cost to come (h(x) + g(x)). We follow the above algorithm step by step to get the desired state.

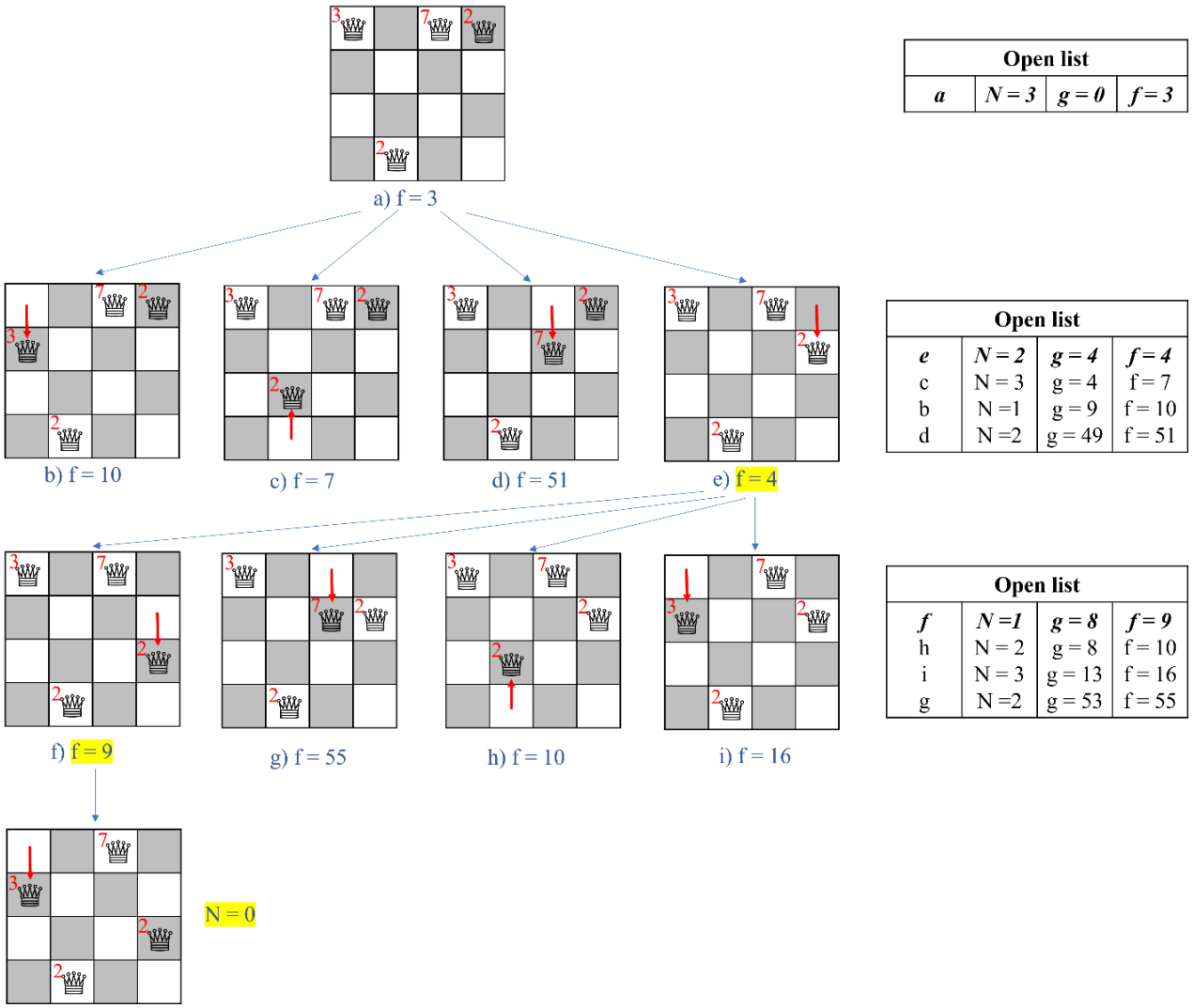


Fig. 3 – Example of 4-Queens problem with A implementation*

The cheapest element from the open list, with the lowest f cost (h(x) + g(x)), is removed, and it is expanded to the neighboring nodes, that is the configuration of the board by moving one queen up and down in the column. For each new configuration, if it hasn't been visited already, the heuristic, cost to come, and total f cost is computed, and the neighboring nodes, is added to the open list.

1.2.4 Solution

The solution for the provided 5x5 initial board was successfully returned with the following output:

Move column 1 down 2 squares.
Move column 2 up 2 squares.
Move column 3 down 1 square.

Elapsed time: 0.06 s
Nodes expanded: 220
Search depth: 5
Solution Cost: 104

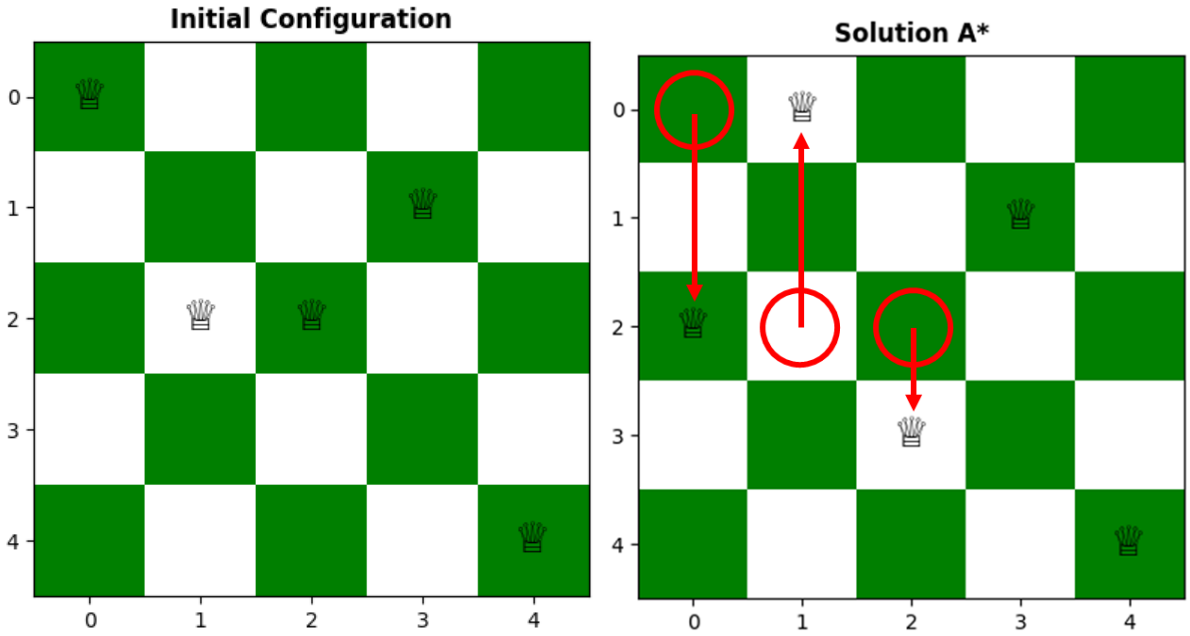


Fig. 4 – Initial board configuration and final solution using A star

Heuristic choice

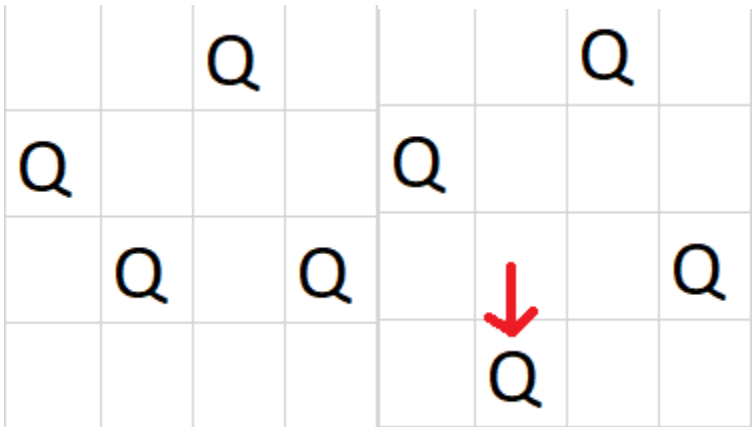
Two Possible Heuristics were explored by us:

Heuristic-1: The heuristic we chose is the number of attacking queens, which is *not admissible*, because it overestimates the actual cost. An admissible heuristic must always *underestimate* the actual path cost to the goal which means that if we can find a position where it overestimates the path cost, we will have proven it to be inadmissible

H(x) = no. Of attacking queens

Consider the example in Fig. 11. The state on the left has heuristic N = 2. Let's assume the weight of all queens is 1. If we make one move and we move the queen in column 2, down 1 square, the number of attacking pairs will be 0. Upon observation, the actual path cost is 1 i.e., (weight of the queen) ^2 * number of moves= 1*1= 1

Hence, with path cost 1, the goal state was achieved. The heuristic, therefore overestimated the path cost as 2, which proves that it is not admissible.



Heuristic-2: We define a more informative heuristic as compared to the above (no. of attacking queens). ***H(x) = no. Of attacking queens * (sum of squares of the weight of the attacking queens)***. The idea behind this heuristic is to disincentivize positions where heavier queens end up attacking each other. This heuristic is not admissible as it is strictly greater than the number of attacking queens, which has already been proven to be an inadmissible heuristic.

Comparing the Performance of Heuristics and the effect of multiplying them by 10:

This comparison was carried out by using both choices of heuristics on 6 random 7*7 and another 6 random 6*6 boards. The results are shown in the table below. Here, 'old heuristic' refers to simply the number of attacking pairs and 'new heuristic' refers to the one where number of attacking pairs is multiplied by the square of the weight of attacking queens

A-star																				
old heuristic*10					new heuristic*10					old heuristic					new heuristic				
size	trials	time	cost	nodes	size	trials	time	cost	nodes	size	trials	time	cost	nodes	size	trials	time	cost	nodes
7*7	1	4.18	292	3570	7*7	1	0.1	398	420	7*7	1	4.27	292	3570	7*7	1	0.09	398	420
	2	20.62	368	8400		2	0.43	380	1050		2	20.78	368	8400		2	1.11	368	1785
	3	could'nt solve				3	0.17	299	630		3	could'nt solve				3	0.17	299	630
	4	could'nt solve				4	1.56	368	2100		4	could'nt solve				4	0.35	368	945
	5	28.85	214	10080		5	1.27	465	1995		5	29.11	214	10080		5	1.25	214	1890
	6	could'nt solve				6	0.23	705	735		6	could'nt solve				6	0.27	705	735
6*6	1	could'nt solve			6*6	1	0.56	611	1248	6*6	1	could'nt solve			6*6	1	1.32	372	2028
	2	0.68	448	1404		2	0.28	557	858		2	0.69	448	1404		2	4.73	557	4056
	3	could'nt solve				3	1.05	338	1872		3	could'nt solve				3	0.83	338	1638
	4	could'nt solve				4	0.03	572	234		4	could'nt solve				4	0.03	572	234
	5	7.05	133	4992		5	5.15	133	4212		5	7.07	133	4992		5	18.86	482	8034
	6	could'nt solve				6	0.16	186	624		6	could'nt solve				6	0.05	186	325

Fig. 5 - Comparison of A for various heuristics*

From the above data, we may draw a few inferences.

1) The new heuristic (Heuristic #2) seems to be significantly more reliable and effective than the other heuristic that does not consider the weight of queens. Its Average time for 7*7 boards is 0.54 seconds whereas the other one often cannot solve the board at all within 30 seconds
2) Multiplying either heuristic by 10 makes the algorithm act greedier. This can be seen in the number of nodes explored in each case for the 7*7 board. For the new heuristic, average nodes explored is 1067

whereas for 10*new heuristic is 980. The average cost, however, for the former is 394 whereas for the latter is higher at 435.8

3) For the 6*6 board, the trend reverses, with the average cost for 10*heuristic being higher. But this may due to a single outlying trial which ended up exploring over 8000 nodes and found a bad solution. If that trial is excluded, the performance of both heuristics is identical. This may indicate that that for smaller boards, the heuristic begins to matter less.

Greedy

old heuristic*10					new heuristic*10					old heuristic					new heuristic				
size	trials	time	cost	nodes	size	trials	time	cost	nodes	size	trials	time	cost	nodes	size	trials	time	cost	nodes
7*7	1	4.18	292	3570	7*7	1	0.1	398	420	7*7	1	4.27	292	3570	7*7	1	1.2	299	1890
	2	3.18	280	2940		2	1.23	299	1890		2	3.21	280	2940		2	1.23	299	1890
	3	0.18	844	630		3	1.71	368	2100		3	0.18	630	844		3	1.7	368	2100
	4	1.16	406	1890		4	1.57	465	1995		4	1.19	406	1890		4	1.57	465	1995
	5	0.48	705	1050		5	0.27	705	735		5	0.47	705	1050		5	0.27	705	735
	6	2	824	1456		6	0.97	576	1522		6	2.16	824	1456		6	0.98	576	1522
6*6	1	0.69	649	1404	6*6	1	0.97	611	1560	6*6	1	0.68	649	1404	6*6	1	0.98	611	1560
	2	0.04	183	312		2	0.32	557	858		2	0.04	183	312		2	0.31	557	858
	3	0.11	338	546		3	0.98	338	1872		3	0.11	338	546		3	0.95	338	1872
	4	0.02	572	234		4	0.03	572	234		4	0.02	572	234		4	0.03	572	234
	5	1.14	482	794		5	4.41	133	3822		5	1.13	482	1794		5	4.38	133	3822
	6	0.45	186	1170		6	0.16	186	624		6	0.46	186	1170		6	0.16	186	624

Fig. 6 - Comparison of Greedy Search for various heuristics

Greedy search was tested with 6 more boards of 7*7 and 6*6 each. Here, it seems like whether the heuristic is multiplied by 10 does not matter much, with the performance for the new heuristic remaining practically the same whether the factor of 10 was included. Of the 6 trials in the 7*7 board, only one differs between the two, giving a slight edge to the heuristic without being multiplied by 10. However, this seems to be a statistical anomaly. Overall, we may conclude that the performance is mostly unaffected when it comes to greedy search.

1.3 Analysis

1.3.1 Time competition

We ran A* and greedy search algorithms to solve different boards in increasing size to see how large a board could be solved in 30 seconds. The initial configuration of the board, as well as the weights were randomized.

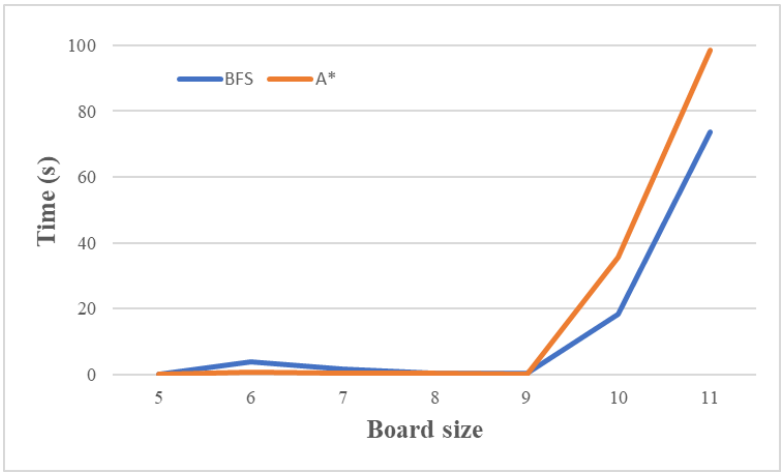


Fig. 7 – Time competition

Both search algorithms are fast for smaller board dimensions. However, as shown in Fig. 7, the solving time remarkably increased for board size greater than 9. Greedy search was able to solve a 10x10 board in less than 30 seconds, however A* took 35.76 seconds for the same board.

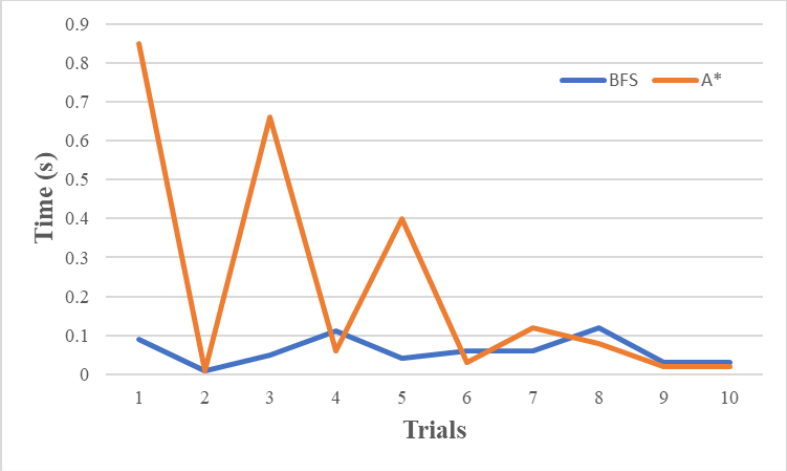


Fig. 8 – Time competition for different 5x5 boards

We run the algorithms for several trials for 5x5 boards with random initial configurations (Fig. 8). The solving time depends on the initial configuration of the board and how close the initial configuration is from the solution: the average and standard deviation for the competition time for Greedy Search was 0.06 ± 0.034 seconds, and 0.225 ± 0.289 seconds for A*. Over trials, the number of explored nodes is always higher for A* than Greedy Search, and A* successfully returns the solution with lowest cost.

1.3.1 Effective Branching factor

The ***effective branching factor B*** is the average number of successor nodes generated by a certain board configuration. It is calculated as follows:

$$B = x^{1/d}$$

where x is the total number of nodes explored and d is the search depth.
The branching factor for greedy search is **2.7,** while the one for A* is ***2.39***. (For given board.csv)

So, we tried a lot of experimentation with different size boards and configurations, and we found that in few cases the effective branching factor for greedy search is more than the effective branching factor for A* and sometimes the opposite. For the given board (board.csv) we got effective branching factor for greedy search as 2.7 because the nodes expanded are 165 and search depth is 4. Whereas, for the A* we got the branching factor as 2.39 because the nodes expanded are 220 and the search depth is 5. We believe that there are few cases where greedy search might give a higher effective branching factor than A* for a particular board. Different sets of configurations may be responsible for different and weird effective branching factors.

1.3.2 Discussion

We have seen how Best-first search and A* differ in terms of the evaluation function for choosing which node to expand first: the latter takes into account the distance to the goal, as well as knowledge about the path travelled so far *(f(x) = g(x) + h(x)*). Greedy Search, on the other hand, only considers the heuristic *(f(x) = h(x)*). As discussed in section 1.3.1, depending on the configuration of the board, A* might take more time. It is an optimal algorithm, which means it will return the solution with the lowest cost. Opposite to that, Greedy Search is not optimal and might run faster. Moreover, the memory required for A* is higher than Greedy Search, because more nodes are explored. Hence, we are trading time and memory for solution quality.

1.3.3 Horizontal Movement

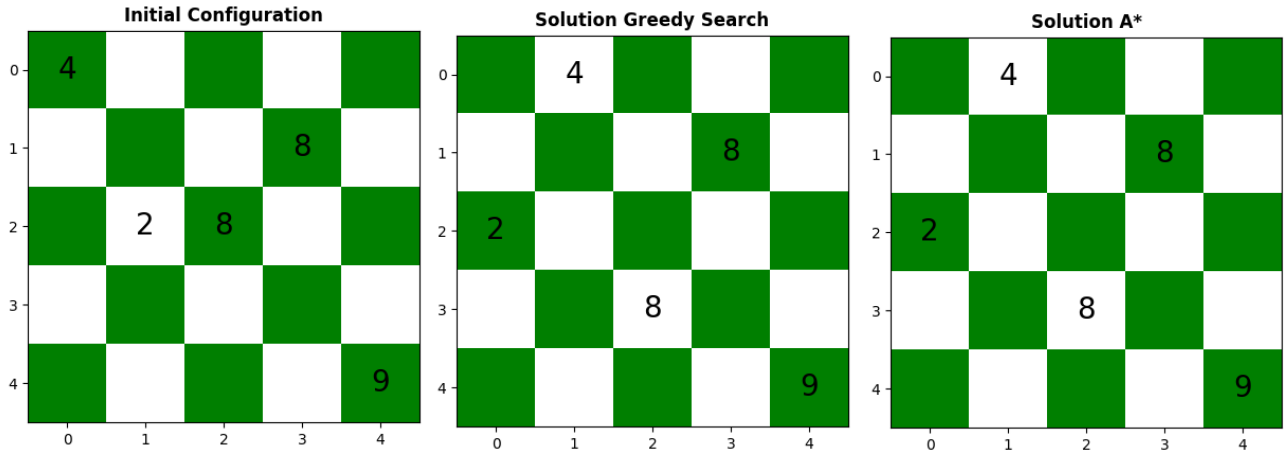
We implemented greedy search and A* algorithms for the N-Queens problem, allowing both vertical and horizontal movements of the queen. At each iteration, we explore the 4 neighbors for a certain queen, that is, given an initial queen position (x, y), its neighbors will be [(x, y+1), (x+1, y), (x, y-1), (x-1, y)].

Given the 5x5 board provided for this assignment, we therefore compared the performance when allowing movements in different directions. In the Table below, V denotes the solution allowing only vertical moves, while H + V stands for horizontal + vertical. The number of expanded nodes for A* is higher, as well as the search depth. However, the solution quality is similar for greedy search.

	GREEDY SEARCH		**A***	
	V	H + V	V	H + V
Elapsed time	0.01	0.04	0.03	0.05
Nodes expanded	165	60	220	100
Search depth	4	4	5	6
Branching factor	2.7	2.78	2.39	2.15

The larger the board size, the higher the competition time and the number of nodes explored for both algorithms with respect to only vertical moves allowed.

The results were the following (the numbers represent the weights of the queens):



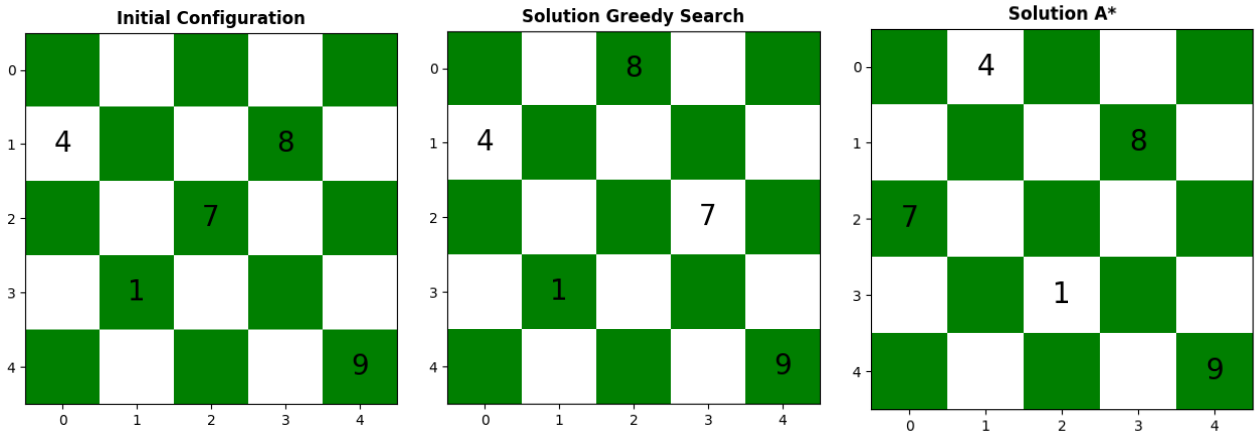


Fig. 9 – Initial board configuration and solution allowing horizonal moves

We compared the performance with different board sizes and plotted the competition time and the search depth. The higher the board size, the more time A* takes with respect to Greedy Best-First search. In general, A* explores more node and has an higher depth search.

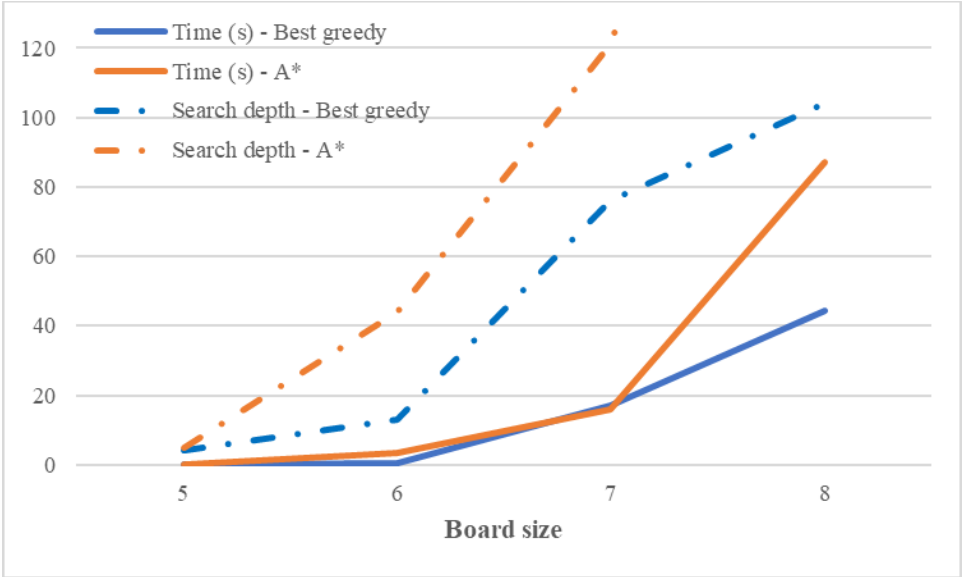


Fig. 10 – Time competition and search depth for different board sizes

Considering the performance in terms of ***solution cost,*** considering both horizontal and vertical moves returns a cheaper solution, when possible. There is thus a tradeoff between time and number of explored nodes and solution cost. Fig. 7 shows the time required to find a solution for different grid sizes, allowing only vertical moves; 10x10 board size could be solved in 40s. However, allowing both horizontal and vertical moves, more than 40 seconds are required to solve an 8x8 board. Since the tree has more nodes (because of considering 4 neighbors instead of 2), it might take more time to find a solution and explore more nodes. However, the solution cost can be improved, especially with A*.

	GREEDY SEARCH		A*	
	V	H + V	V	H + V
Solution Cost (Configuration 1)	209	20	104	20
Solution Cost (Configuration 2)	162	192	162	131
Solution Cost (Configuration 3)	137	117	25	17

2. Part 2 – Local Search

The second part of the assignment will focus on solving the N-queens problem with the local search algorithm of Hill Climbing with Simulated Annealing. The reason simulated annealing is implemented is that hill climbing tends to get stuck at local optima (not the global optimum) and fails to explore further. Simulated annealing helps in overcoming this problem by injecting a degree of randomness into the search, allowing it to increase the comprehensiveness of its exploration. The main difference between local and path search is that local search does not use a priority queue and only compares the neighbors of its current state to select its next move. This affords it the advantage of requiring less memory with the downside of tending to repeat moves.

2.1.1 Hill Climbing with local search

Hill Climbing is also a heuristic-driven algorithm which simply chooses the best available neighbour and thus it gets its name hill climbing. The algorithm is a type of local search as it sees only the immediate neighbors. Since it does not backtrack the search space, it does not have memory and it is also less time consuming. It ends when the value of all neighbors is less optimal than its current state and gives that state as the solution for the problem.
The major drawback of hill climbing is that it may not find the global maximum for the given problem and get stuck at a point when the neighboring nodes do not have an improved value.

1. The algorithm will start with a single random position, and does random restart from the same position, if the initial position is the solution, then return success and end.
2. Iterate this process until a solution is found, if that is the final solution, then return success and quit
3. Exit

2.2 Simulated Annealing

Simulated annealing is an optimization algorithm that uses probabilistic random exploration. This algorithm can be used when vanilla hill climbing gets stuck in a local maximum. The term ‘annealing’ refers to the metallurgical process where a metal is constantly heated and involves cooling in a controlled manner to alter its material strength. Likewise in AI, Simulated annealing has a temperature value, cooling rate, Boltzmann's constant. The advantage of using simulated annealing is that it also explores its nearest neighbors by the given formula below:

$$P = 1 \quad \text{if } \Delta c \leq 0 \qquad \text{(or)} \qquad e^{-\frac{\Delta c}{t}} \qquad \text{if } \Delta c > 0,$$

where P is the probability; Delta c is the change in cost and t is the current temperature

The calculated P from this formula will be used to accept the new solution. When the value of T is high, the probability is also high which leads to more random exploration and as it decreases over time, the search acts more like hill climbing.

2.2.1 Algorithm and pseudocode (Assuming that the state variable is minimized)

1. The algorithm will start from a random position and check if it is the solution or not.
2. Unlike hill climbing this will also look at the worst neighbors and the better ones too.
3. If it is better than the current state, the p=1 and current will become the neighbor.
4. If it is not better then also the simulated annealing runs a probability and finds whether the worst neighbors are suitable.
5. Iterate the process from step 3, to get the final optimal solution.

```
While Temp> Min Temp and current state != goal state
        Find set of all neighbors = []
        Evaluate state value for each neighbour
        If neighbor's state < current state
Current state=neighbour
        Else find probability
Choose a random number between 0 and 1
If prob>random number
current state=neighbour
        Decrease temperature according to cooling schedule
        Check if current state=goal
```

2.2.3 Implementation of Simulated Annealing with Hill climbing

We will take the initial board of the weighted n-queens problem with random configuration and check whether it is the final solution. If not, then with the input temperature values T, k, and the cooling rate (cr) we will check every possible move on the board. If the current state is worse than the neighbor, then the neighbor becomes current. If a neighbor is not better than the current value, then algorithm uses a probabilistic approach and sees whether the neighbor is better than the current. We will also see whether the given board can be solved within 30 seconds. The total cost for the solution, the number of moves, and the restarts are calculated.
The function being used to evaluate the states is a weighted sum of path cost ($Mass^2$*distance) and the number of attacking pairs. The weight of path cost is taken to be 1 and that of attacking pairs is calculated as 25*n/(1.1*(n-1)). This value was obtained by calculating the average number of attacking pairs on a random n*n board for various values of n, with 1 queen in each column; using linear regression to establish a relationship between them and then equating it to the average path cost. This ensures that the values of both are always comparable, irrespective of the size of the board. The average values were found statistically by generating 10000 random boards of various sizes.

Board Size	Avg Attacking Pairs
3	2.1
4	3.3
5	4.4
6	5.5
7	6.7
8	7.8
50	56

2.2.4 Solution

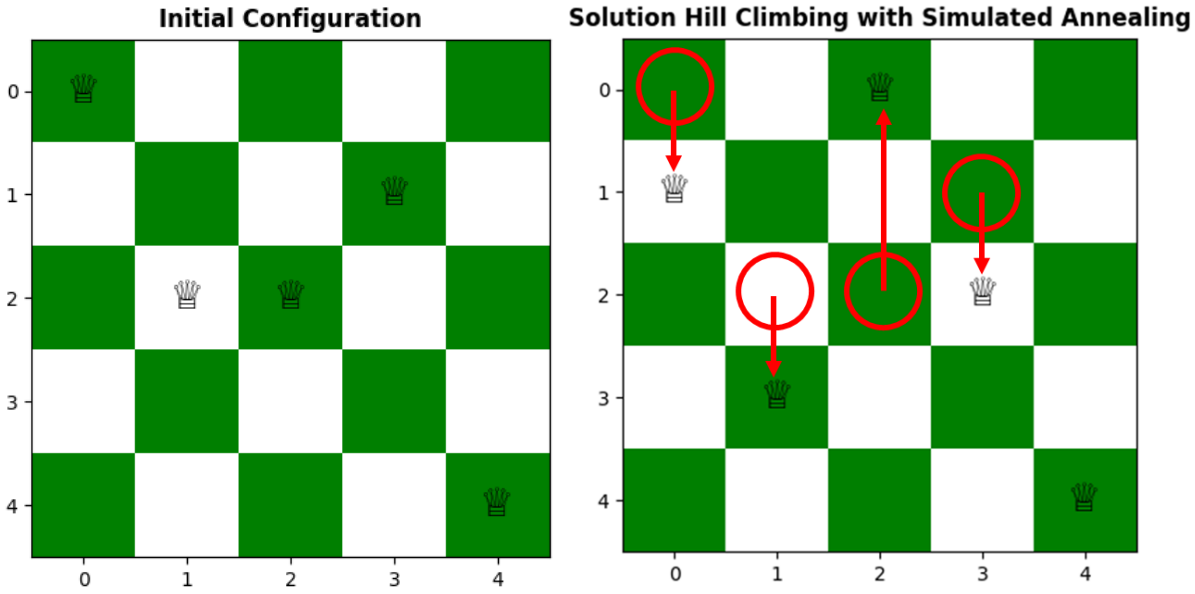


Fig. 11 – Simulated annealing initial board and solution

Elapsed time: 0.31 s
Move column 1 down 1 square.
Move column 2 down 1 square.
Move column 3 up 2 squares.
Move column 4 down 1 square.
The solution cost is: 212
Nodes expanded: 61

2.2.5 Analysis

The heuristic used for Hill Climbing is similar to the first part, apart from the introduction of a weighted sum. Initially, while testing the code, a naïve unweighted sum was chosen but the weighted sum significantly outperforms the former. This is likely because the absolute magnitudes of path cost and attacking pairs are vastly different (the latter is much smaller on average) which causes the algorithm to overly prioritize low-cost moves even if they do not decrease the number of attacking pairs. And without a priority queue or memory as is present in A*, this sort of

movement inevitably results in loops as the temp decreases. While the weighted sum can solve up to 10x10 boards reliably, unweighted sum fails at 5x5.

The chosen annealing schedule is *geometric decay*. This is because it was seen from testing that there was not a significant difference in the effectiveness of various cooling schedules (the tested ones include harmonic decay i.e. $T = \frac{T_i}{1+n}$ and Logarithmic Decay i.e. $T = \frac{T_i}{1+\log(n)}$ apart from geometric decay) but sometimes, the geometric decay with a sufficiently low cutoff temperature was able to solve larger boards of size 12 (although this took longer than 30 seconds).
As for the initial temperature, its value was chosen by trial and error and was different for every board size. Feasible values were in the range of 10-1000 with smaller values failing on larger boards. Keeping the cooling rate constant, high temperatures correspond to more exploitation and fewer restarts.
The effect of temperature on the number of nodes explored (moves made) and the randomness of exploration can be seen below for a 6x6 and 8x8 board.

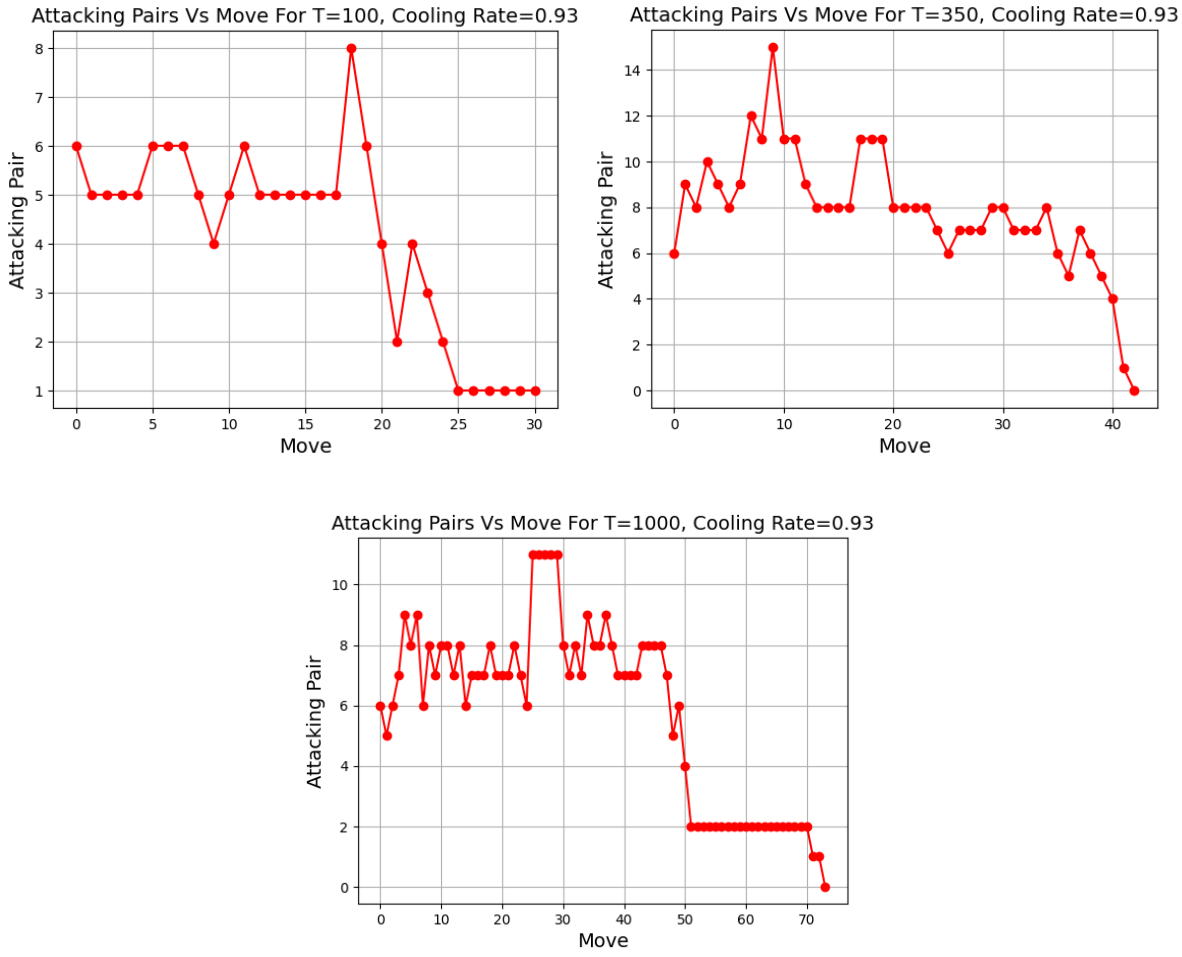


Fig. 12 – Attacking pairs vs moves for a 6x6 board

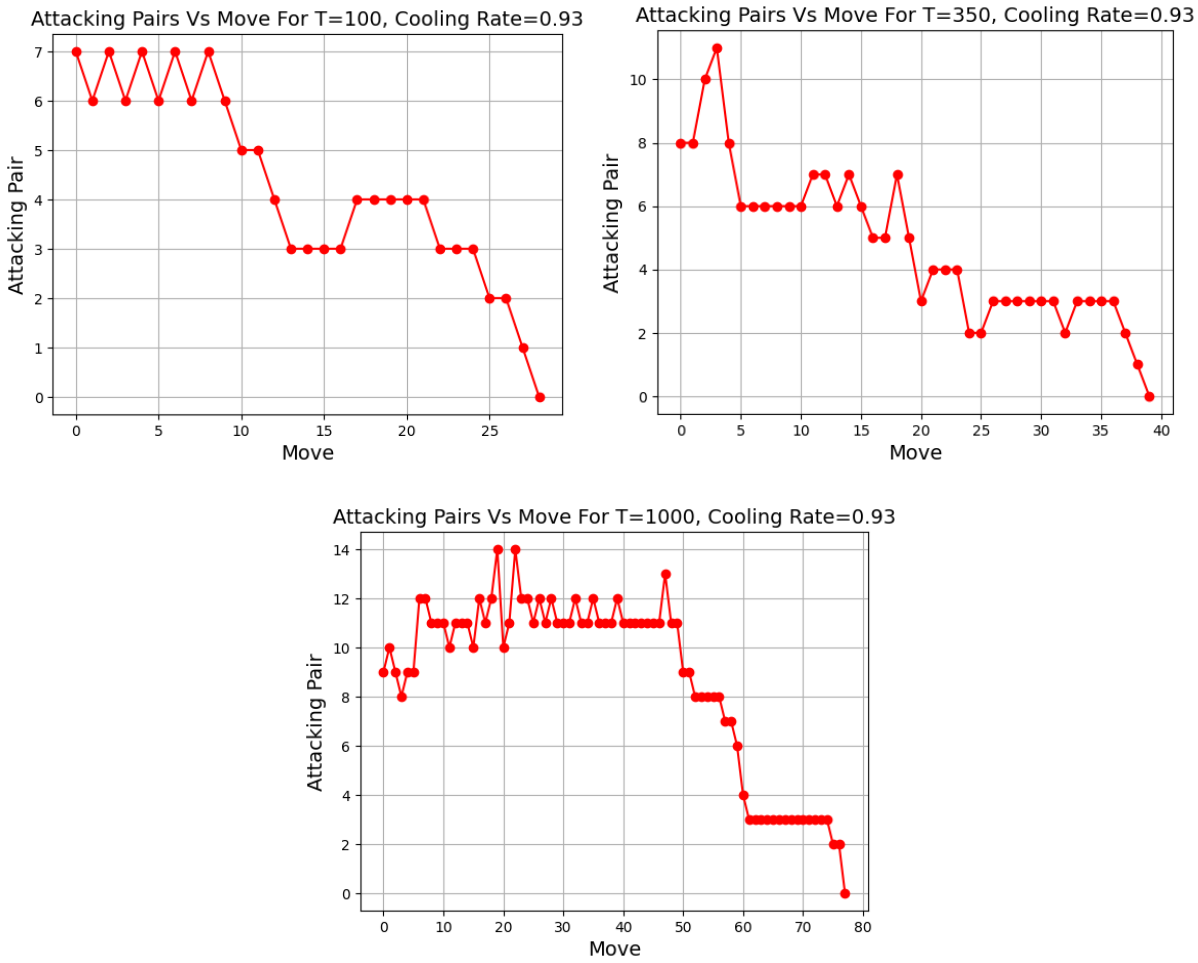


Fig. 13 – Attacking pairs vs moves for an 8x8 board

Local search was more or less as effective as Path Search but consumed more time. The memory usage, however, was low. Boards up to 13x13 could be solved at least sometimes with values up to 9x9 being reliable. A graph based on a sample size of 20 can be seen below:

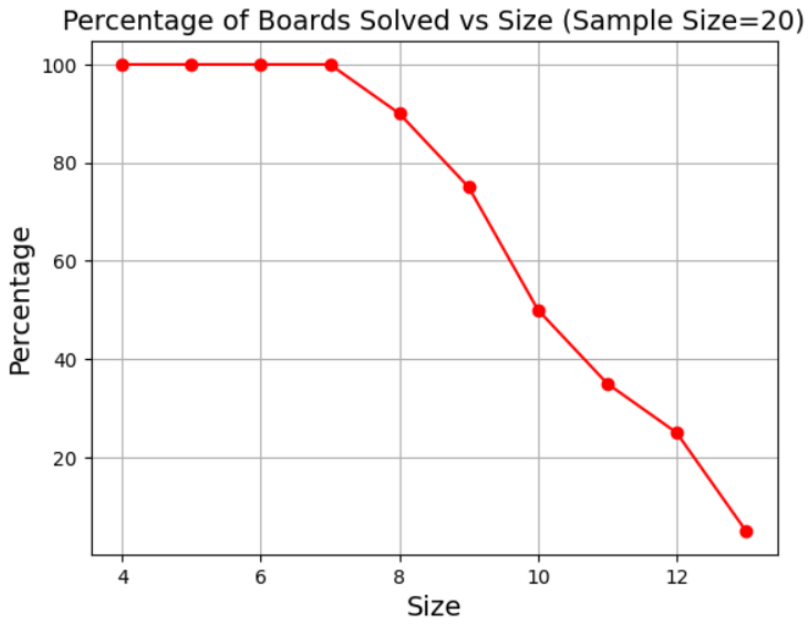


Fig. 14 – Percentage of solved boards vs Size

Horizontal Moves in Local Search:

Enabling horizontal moves makes it more difficult to solve larger boards (particularly those of size 9x9 and above) but the effect on smaller boards is negligible.

Below is a table showing results from a comparison of enabling and disabling horizontal moves for a specific 8*8 board that was generated randomly.

	Horizontal and Vertical				Vertical		
Trial No.	Iterations	Moves	Cost		Iterations	Moves	Cost
1	11	48	3115		3	35	2559
2	3	47	2674		1	14	1022
3	3	44	1849		2	36	2581
4	12	51	2720		4	35	2842
5	5	39	2152		5	32	1991
6	20	44	1904		1	45	2231
7	1	41	1357		10	41	2589
8	6	47	2244		8	42	2236
9	28	58	2208		7	38	1770
10	12	49	2229		9	32	1797
Average	10.1	46.8	2245.2		5	35	2161.8

Fig. 13

In the above table, Iterations, refers to the number of times the annealing algorithm had to restart before a solution was found, Moves are the number of moves needed to arrive at the solution and cost refers to the total cost of making said moves

The above data shows that enabling horizontal moves decreases the performance of the algorithm.

This was further cemented by a second test which involved allowing the algorithms to run for 30 seconds and analyzing their output:

Reference Board:

0	0	3	0	0	8	0	0
0	6	0	0	6	0	0	0
0	0	0	0	0	0	0	0
0	0	0	7	0	0	0	9
8	0	0	0	0	0	0	0
0	0	0	0	0	0	0	0
0	0	0	0	0	0	2	0
0	0	0	0	0	0	0	0

Best Result after Allowing Horizontal Moves (Cost=2555, No of Solutions found in 30 seconds=4):

0	6	0	0	0	0	0	0
0	0	0	0	0	0	3	0
0	0	0	0	2	0	0	0
0	0	0	0	0	0	0	9
8	0	0	0	0	0	0	0
0	0	0	7	0	0	0	0
0	0	0	0	0	8	0	0
0	0	6	0	0	0	0	0

Best Result While staying restricted to Vertical Moves (Cost =1040, No of Solutions found in 30 seconds=11):

0	0	3	0	0	0	0	0
0	0	0	0	6	0	0	0
0	6	0	0	0	0	0	0
0	0	0	0	0	0	0	9
0	0	0	0	0	8	0	0
0	0	0	7	0	0	0	0
0	0	0	0	0	0	2	0
8	0	0	0	0	0	0	0

Thus, staying restricted to vertical moves allowed for a lot more solutions and the best solution was less than half the cost of the solution when horizontal moves are enabled.

This may indicate that the increase in size of the search space, despite opening up the possibility of better solutions, adds too many suboptimal paths for the search algorithm to filter out.

Restarts vs Annealing:

To analyse the utility of including restarts, a test was conducted, involving 10 random 8*8 and 6*6 boards. For each random board, Annealing was run using 1, 5, 10, 15, 20, 40 and 100 restarts within 30 seconds and the best result from these restarts was stored.

The results are shown in the table below:

Restarts	Att. Pairs Left	Cost	Restarts	Att. Pairs Left	Cost	Restarts	Att. Pairs Left	Cost	Restarts	Att. Pairs Left	Cost	Restarts	Att. Pairs Left	Cost		Restarts	Att. Pairs Left	Cost		Restarts	Att. Pairs Left	Cost
1			**5**			**10**			**15**			**20**				**40**				**100**		
8*8																						
1	2	1811	1	2094		1	2968		0	2534		0	3713			0	2424			1	1834	
2	1	1982	0	1792		1	1305		0	2104		0	1600			1	1927			1	1622	
3	2	2297	0	2034		1	1792		0	2405		0	2143			0	1991			2	1443	
4	3	1860	1	2313		1	2852		0	2069		0	2209			0	1875			1	1496	
5	1	3036	1	2759		0	2674		1	2566		1	2504			1	3284			1	1912	
6	0	2538	0	2122		0	2022		0	2556		0	1883			0	2324			3	1678	
7	1	2483	0	1285		0	2071		0	2181		1	641			0	1417			2	2439	
8	1	2938	1	2876		0	1790		0	2601		0	2800			0	2478			2	1613	
9	1	2406	2	3566		1	1976		0	2711		0	2340			0	3314			1	1734	
10	1	3559	1	3358		1	3466		1	3522		1	2409			1	3177			3	1888	
Average:	1.3	2491	0.7	2419.9		0.6	2291.6		0.2	2524.9		0.3	2224.2			0.3	2421.1			1.7	1765.9	
6*6																						
1	1	1093	0	1849		0	2120		0	1724		0	1795			0	1704			0	2080	
2	1	3331	0	2025		0	2053		1	2469		1	2153			0	2368			0	1338	
3	2	794	0	2379		1	2736		0	2485		0	2550			0	2577			0	2299	
4	2	458	1	862		0	2390		0	1004		0	2999			0	1635			0	4201	
5	1	2144	1	2348		0	2571		0	1925		1	2661			0	1839			0	2413	
6	0	875	1	1046		0	845		0	799		0	841			0	987			0	669	
7	1	1732	1	1574		1	1066		0	2350		1	1066			0	2368			0	2548	
8	1	1045	1	429		0	1811		1	772		1	1589			0	1917			0	1523	
9	1	1307	0	2002		0	2068		1	1190		1	1967			1	2104			0	1596	
10	2	628	1	2088		0	2306		0	2130		0	3223			0	3165			0	2739	
Average:	1.2	1340.7	0.6	1660.2		0.2	1996.6		0.3	1684.8		0.5	2084.4			0.1	2066.4			0	2140.6	

From these results, we can infer that smaller boards tend to be better solved by taking more restarts. Consider that for the 8*8 board, 100 restarts had the worst performance in the set (based on average number of left-over attacking pairs after 30 seconds=1.7. Possibly, this means that 30/100=0.3 seconds per annealing cycle is too low for an 8*8 board) whereas it was the best performing for 6*6 (average left-over attacking pairs are 0). For both boards, focusing on a single run of annealing performed terribly (worst-adjacent in 8*8 and the worst in 6*6). Hence, we may conclude that a healthy number of restarts can greatly improve the performance of annealing, and, the number of restarts, for a given amount of time should be higher for smaller search spaces. Of course, the performance of any board size does not increase indefinitely with the number of restarts, as can be seen in the 8*8 board where 40 performed well but 100 was a relative failure. Thus, a certain window of an effective number of restarts exists for every search space and this number is usually >1.