K - Knights & Q - Queens Problem

Mukund Sharma

Student, MS CS, George Mason University, Fairfax (msharm@gmu.edu)(G01374620)

ABSTRACT

This Paper provides a brief description of how to solve the K Knights and N Queen problem by making use of AI algorithms. The algorithms discussed further are Hill Climbing (HC), Simulated Annealing (SA) Algorithm, and Local Beam (LB). The three algorithms are written, implemented, and are further compared among themselves to reveal which is faster for solving the problem.

1. INTRODUCTION:

A chessboard of size N X M must be filled with Q Queens and K Knights, and the problem specifies four conditions: a queen cannot attack another queen, a queen can attack a knight, a knight cannot attack another knight, and a knight cannot attack a queen. We may categorize functions into polynomial and exponential categories, and then further split computing issues into tractable and intractable categories. A issue's tractability is determined by how difficult it is in relation to how long it takes to solve the problem. If the code ran a solution for the problem in a small amount of time, then it can be solved in polynomial time and will be named under tractable problem. It is directly linked with time complexity. But, there are a few problems, which can only be solved by some given algorithms, whose compile time grows as quickly as the input size grows and these problems cannot be solved in polynomial time. These problems are referred to as intractable.

A difficult issue that is frequently brought up while talking about different searching issues is the "Q" Queens and "K" Knights Problem. In order to prevent any two queens or knights from endangering one another or a queen from being threatened by a knight, the objective is to position N number of queens and K number of knights on the (N*M) matrix.

2. ALGORITHMS IMPLEMENTED:

2.1 Hill Climbing(HC):

The hill-climbing search algorithm is a technique that allows moving uphill for generating output. It gets terminated on reaching the peak value where it cannot find any new neighbor with a higher value. The algorithm is implemented in a way such that there are 3 states for knights: state, neighbourState, opState, and there are 3 states for queens stateQ, neighbourStateQ, opStateQ. And correspondingly there are 3 boards Board, NeighbourBoard, opBoard. State, StateQ and Board are the initial and will be the final states since every time an optimized solution is received then it gets stored in these state boards. The neighbourState, neighbourStateQ and neighbourBoard as the name suggest are the corresponding neighbors of our original state. The opState, the opStateQ and the opBoard is the state where relative local maxima are stored and will be assigned to original Board if is better than current original.

The data structure for the current node just has to store the state and the value of the objective function because the method does not maintain a search tree. Hill climbing doesn't look beyond the current state's near neighbors. It merely keeps a record of its current state and close neighbors.

```
function HILL-CLIMBING( problem) returns a state that is a local optimum
   inputs: problem, a problem
   local variables: current, a node
                    neighbor, a node

   current ← MAKE-NODE(INITIAL-STATE[problem])
   loop do
      better ← false
        for each neighbor of current
            if VALUE[neighbor] > VALUE[current]
               current ← neighbor
               better ← true
             break
         end
    end
     if better = false then return [current]
   end
```

2.2 SIMULATED ANNEALING(SA):

Algorithms like SA are used to stochastically optimize the conditions for the global search. The method was modeled after the metallurgical procedure of annealing, which involves rapidly heating metal to a high temperature and then gently cooling it. It gains strength as a result, which also makes dealing with it easier. At first, the material's atoms are stimulated at a high temperature, which permits them to move about a lot. The annealing procedure functions in this manner. The atoms are then given time to progressively lower their excitement as they transition into a new, more stable configuration.

The implementation of SA algorithm is similar to that of hill climbing but it adds a factor for probability and temperature which helps the algorithm to even check the weak neighbours. One of the most popular heuristics approaches for resolving optimization issues is the SA algorithm. The term "annealing technique" refers to cooling metals gradually after high heat treatment while defining the ideal molecular configurations of metal particles where the mass's potential energy is reduced. In general, the SA algorithm utilizes an iterative motion based on a changeable temperature parameter that mimics the metals' annealing process.

```
function SIMULATED-ANNEALING(problem, schedule) returns a solution state
       inputs: problem, a problem
               schedule, a mapping from time to "temperature"

       current ← MAKE-NODE(problem.INITIAL-STATE)
        for t = 1 to ∞ do
            T ← schedule(t)
```

```
if T = 0 then return current
 next ← a randomly selected successor of current
 ΔE ← next.VALUE – current.VALUE
 if ΔE > 0 then current ← next
 else current ← next only with probability eΔE/T
```

2.3 LOCAL BEAM SEARCH (LBS):

It would seem like a drastic solution to the memory problem to keep just one node in memory. Instead of simply one state, the local beam search algorithm keeps track of k states. There are k randomly created states at the start. All of the k states' successors are formed at each phase. The algorithm comes to a stop if someone is a goal. In any other case, it repeats after choosing the k best successors from the entire list. A local beam search with k states can initially appear to be nothing more than performing k random restarts concurrently rather than sequentially. In actuality, the two algorithms diverge significantly. Each search process operates independently of the others in a random-restart search. The simultaneous search threads in a local beam search exchange helpful information. The states that produce the finest successors are, in essence. The algorithm swiftly gives up on fruitless searches and directs its energies to areas where there is the greatest chance of success.

1. function localBeamSearch(problem, k) returns a solution state
2. start with k randomly generated states
3. loop
4. generate all successors of all k states
5. **if** any of them is a solution then **return** it
6. **else** select the k best successors

3. OBSERVATIONS:

N	M	Q	K	Tmax(s)	HC(s)	SA(s)	Other(LBS)(s)
5	5	5	0	120	7.8	14	2.3
5	5	0	5	120	3.2	2.2	2
5	5	5	5	120	2.7	8.4	2
10	10	10	0	120	10	30	2
10	10	0	10	120	1.6	1.5	1.3
10	10	10	10	120	4	2	1.5
15	15	15	0	120	8	60	2
15	15	0	15	120	1.5	1.5	1.3

15	15	15	15	120	2.2	1.6	1.6
20	20	20	0	120	60	30	2
20	20	0	20	120	2	1.8	1.9
20	20	20	20	120	1.7	1.9	1.9

4. CONCLUSION

After performing numerous tests on the code, it can be deduced that the best approach would be simulated annealing if the number of conflicts is a concern and must be absolutely zero, but if the user wants the output faster and the number of conflicts is not a major concern, the local beam technique works flawlessly. Both methods provide faultless results, and as we continue to learn more and increase our knowledge, we will be able to modify and enhance the code even more.

5. REFERENCES

1. https://zoo.cs.yale.edu/classes/cs470/materials/aima2010.pdf
2. https://www.geeksforgeeks.org/n-queen-problem-local-search-using-hill-climbing-with-random-neighbour/