Program documentation for using Hill climbing algorithms to solve N-queen problem

Vinko Lovrenčić
Mentor: prof. dr. sc. Luka Grubišić
Prirodoslovno-matematički fakultet, Matematički odsjek

February 25th 2021.

Contents

1 General information, common code

1.1 General information

Here we are going to explain the code used in this project and it's uses, we are gonna explain classes, variables, methods etc. We are not gonna analyze said code/algorithms or compare them, this is something we will be saving for User documentation.
First of all, every program is written in C++, using object oriented programming and other modern programming tecniques. Every program is successfully compiled and tested on gcc version 10.2.1 20201125 (Red Hat 10.2.1-9) (GCC) compiler, on Fedora 33 (Workstation edition) linux system: but it sould work on all OSes (i don't see a reason why it wouldn't work since none of the libraries or functions are OS specific) and most compilers (albeit with possible small changes to the code, some parts of code could be compiler specific). What's also important is that the code is written in C++11 standard, so the compiler might have to be told to use it (most compilers today have it as default. My approach to the problem was to write a class containing a solution to the problem and all functions needed to manipulate solutions, which is contained in nqcommon.h and the just write different main function using that header file as a baseline and using different algorithms to solve the problem.

1.2 Common code

Everything is nqcommon.h is shared among all algorithms and it's a baseline for all of them.
Let's analyze nqcommon.h.
First we define some constants, LEFT and RIGHT: this is used to discaonnect the user programs from the implementation: the algorithm using the header only needd to think its moving the queen left or right, and not HOW to move it left or right.
Next we seed out random numbers using built in random device and using Mersenne Twister pseudorandom generator to generate numbers out of that random device. There are also 2 templates that give as uniformly distributed random number in a given range: one gives integers (**short**, **int**, **long int**...), other gives real numbers (**float**, **double**). Thiese templates were taken from `https://stackoverflow.com/questions/288739/generate-random-numbers-uniformly-over-an-entire-range`.
The class solution is the main part of this file: every other thing it the file is actually part of it. So, let's analyze it:
Private (you can't use thiese outside of the class):

- queen_col is a vector of integers: basically a C++ take on classic C-like arrays. It's is dynamic, fast and a lot less error prone than naked arrays. It dynamically rescales it's size, each rescale taking more memory in one go, so the more we change it the less it will need to resize (something similar is used when deciding how much disk space sould database files take). Solution will be written in a way that every element of the vector represents a row, and number in it represents a column, eg. queen_col[1]=2 means that in row 1 (2nd one, we start from 0) there is a queen in column 2 (third one)

- N is an integer which holds the amount of elements in queen_col: it can be calculated with queen_col.size() but we will avoid calling the same function multiple times if not necessary

Public (every part of the program can use them):

- soution() - basic no input constructor, so we can do stuff like solution A;
- solution(const solution& original) - copy constructor, allows us to do stuff like solution A=B;
- solution(int number) - custom constructor: if we give it a number is automatically consider it N and also gives us a random solution with queen_col's constraints
- operator=(std::vector<int> &input) rewrites an association operator = so we can associate solution class with a vector, we can do stuff like this: (solution) A = (vector<int>) B;
- cost() - function that returns the cost of the current solution, in how many attacks happen
- randomize() - randomizes the solution among all allowed ones
- random_neighbour(int max_step) - gives us a random neighbour of the current solution: picks one random queen and moves it between 1 and *max_step* spaces left or right
- neighbour(int row, int move, int step) - gives us a specific neighbour: the one that is in row *row*, taking that queen and moving it in direction move by *step* steps. if we overflow or underflow (leave the board) we just wrap around.
- print() prints the solution on screen

In this way the variables itself are encapsulated: no one can actually change them manually, only by calling public functions.
Also, this way everything specific about n-queens is in this header: cpps are not gonna be n-queen specific at all, it would be easy to write another header for a different problem and simply use same mains (cpps).

2 Algorithms

Each algorithm cpp has some things in common:

- We take 2 arguments from the command line: first is N, the number of fields and the second one is the maximum allowed step distance. (eg. simple_hill_climb 8 4 will do simple hill climb for 8 queens with maximum allowed step of 4)

- We make the initial solution random with solution current_best(N); and calculate its cost with int best_cost=current_best.cost();
- In the end we print out how good is our solution (minimum cost).

All of the algorithms are different in while loop or some have custom constants. Cpps are named after their algorithm.

2.1 Simple hill climbing

Here we do a while loop until we either find the solution (cost=0) or we can't find a step to climb.
In the loop we simply look at neighbours in order ($row \rightarrow ascendingstepamount \rightarrow LEFT, RIGHT$) and as soon as we find a step that give a better cost then our curren solution we take it.

2.2 Steepest hill climbing

We also do the while loop until we find either a solution or we can't find an improvement.
The difference here is that we look at all possible neighbours and pick the one that will give us the highest cost decrease. Then we take that step.

2.3 Random neighbour hill climbing

This is a variant of steepest hill climbing algorithm. We do the while loop until we find the global optima (will happen, eventually, cause we know it exists for N>3). In the loop we do basic steepest hill climb, but if we can't find a better solution and we are not in global optima we take a random neighbour and use it as next solution, hoping to escape the shouder or local optima.
We use boolean changed to track if we found a better step then current position.

2.4 Random restart hill climbing

A variant of simple hill climbing.
We basically do simple hill climbing until we find the optima, each time starting it from a random position and then comparing the result to top_best - a variable where we save the best solution of all restarts.

2.5 Stochastic hill climbing

Hill climbing based on random climbing.
We choose a random neighbour, and decide whether we want to take that step or try another.
The formula we use is $P(x) = 1/(1 + e^{tc-bc}$ where bc(best_cost) is cost of our current solution, and tc(temp_cost) is the cost of new solution. This is a sigmoid activation function and it suits bias towards better solutions the best. (It is often used in neural networks, for similar reasons)

2.6 Simulated anneling

Here we have multiple constants: INITIAL_TEMP initial entropy we start with, TEMP_DECAY how fast are we losing entrophy, COOLED when do we consider our solution "cooled" enough, BC Bolzmann constant.
How it works is we have a loop from INITIAL_TEMP down to COOLED with TEMP_DECAY step, and in each step we look at random neighbour of our current solution: if its better take the step, if it isn't take it anyways with probability $P(x) = e^{(tc-bc)/(temp*BC)}$ where tc(ran_cost) is the cost of the step we are thinking of taking and bc(best_cost) is the cost of our current solution. Temp is just a loop iterator.
Once our temp reaches COOLED we are done.

2.7 Testing algorithms (testing_algos)

This is a testing function containing all algorithms is a trimmed form (no prints to stdout because those take a lot of time, and no comments) and time constrained (every algorithm is terminated if it's execution time exeeds the predefined constant). It takes 3 inputs from the terminal: N (number of queens), max_step (maximum allowed step) and tries (how many different random boards are algorithms going to solve). (eg. testing_algos 10 10 1000 will give all algorithms 1000 random boards to solve with 10 queens and 10 moves allowed in one state change). This algorithm (along with anneling constants) has it's own constant TIME_CONSTRAINT which dictates maximum allowed time an algorithm can run (in milliseconds).
Also, stochastic, random neighbour and simulated anneling were changed to output the best solution cost they went through, instead of the current one they are working on, because they can actually worsen their current solution at any point hoping to get a better one later.