Artificial Intelligence 1
Lab 2

Name1 (s number 1) & Name2 (s number 2) & Name3 (s number 3)
Team name
Learning Community

May 18, 2018

Exercise 1

1.1 - Hill Climbing

1. We ran the program multiple times. The results are for 8 and 50 queens are in the appendix at the end of the document. We notice that, the larger the grid, the fewer queens are under attack. However, the program almost never solves the problem, but often gets close.

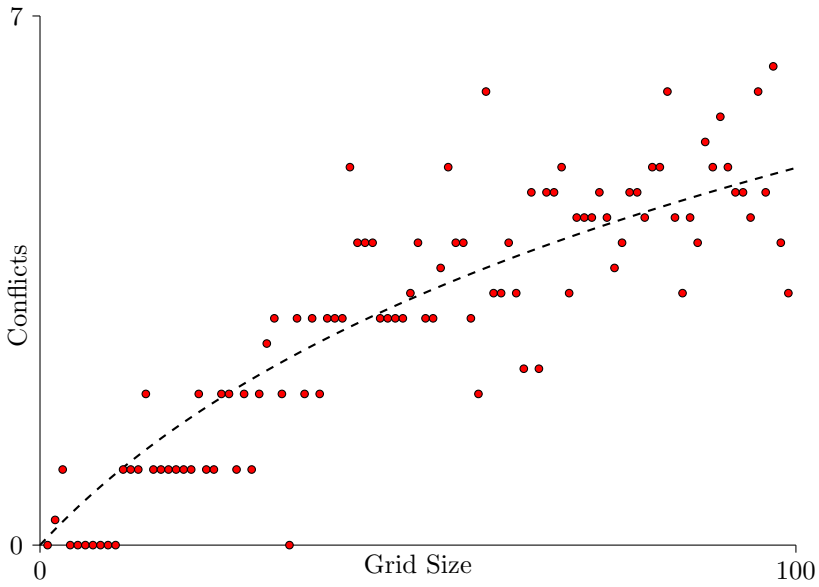
2. The algorithm loops over all queens, and puts them each in the most favourable position. This way, it might look like the solution is going to be optimal in the end, but the pseudo-optimal placement of the first few queens might hinder the yet-to-be-placed queens. Therefore, if this situation occurs, the found 'solution' might not actually be the optimal one, thus the program fails.

3. We might improve the algorithm by implementing a so called Random-Restart-Hill-Climbing algorithm. The idea behind this is as follows. If we dont find the solution, we try again, from another random point. The problem with this approach regarding this problem, is that, there are a lot of random restarts possible. Therefore, we must set a cap to the amount of restarts we allow, say, 25.
   We ran a test, and immediately got a solution:

   ```
   Final State
   .....q..
   ..q.....
   ....q...
   .......q
   q.......
   ...q....
   .q......
   ......q.
   ```

   This is plenty of confirmation that this is a viable solution.

4. We use a bash script to run the program 3 times for every possible value for `N`. We can then graph the average of the results for every grid size (along the horizontal axis).



1.2 - Simulated Annealing

1.3 - Genetic Algorithm

Considerations

There can be much variation in genetic algorithms. For instance, one could change the generation size, the maximum amount of generations or even whole algorithms involved. By trying more combinations, you could discover that for that specific problem, one modification might benefit results.
In our case, we noticed that generation size above a certain threshold does not seem to influence the result anymore. We noticed the same when we modified the maximum amount of generations we allowed. We also implemented two different mutation functions, which can be toggled using a macro.

Design

We tried to make the algorithm as versatile as possible, so we can easily run tests. Therefore, we included macros that control the before named thresholds. Also, by adding command line arguments, we can easily run tests by changing little things.
For the algorithm itself, we used two mutation functions. One where every gene of a chromosome has a chance to be mutated, and one where there is a chance of mutating a single random gene. The last one makes mutation rarer, and therefore less influential. This is benefitial when a generation of chromosomes is already close to convergence.
We also applied a little math to the randomness of the crossover function. We do generate each child as a crossover product of 2 parents, then apply mutation. However, when choosing the parents, we do not use the fitness function as a probability function. We apply a little math to give the 'good' chromosomes a much better chance to be selected than the 'bad' ones.
We did the following. We devide each chromosome's fitness (we will refer to this as f) by a value depending on its fitness (y). We want y to be greater (say the size of the board, y_0) for

lower values of f, and 1 for the maximum fitness (f_m). We can define a linear function

$$y = \frac{1 - y_0}{f_m} f + y_0$$

Since we know the maximum amount of the fitness is the same as the maximum amount of conflicts (see: `evaluateState()`), we can say that $f_m = \frac{y_0(y_0-1)}{2}$, and substitute

$$y = \frac{1 - y_0}{\frac{y_0(y_0-1)}{2}} f + y_0 = -\frac{2}{y_0} f + y_0$$

We can then find p (probability), and simplify to speed up the calculation by the computer

$$p = \frac{f}{y} = \frac{f}{-\frac{2}{y_0} f + y_0} = \frac{f y_0}{-2f + y_0^2}$$

Evaluation

For board sizes up to 10, the genetic algorithm solves the problem consistently. When going higher, it finds solutions which are often more than 90% correct. However, since it does not converge early because it found the solution, the time to terminate becomes exponentially larger, but remains managable up to a board size of about 50.
To conclude, we can say that this algorithm is best to find a solution that is close to correct. Meaning to say, that it does not always find the global optima, but tries to seek out and compare the best local optima.

Exercise 2

Your answers, with explanations, to Exercise 2 go here

Exercise n

Your answers, with explanations, to Exercise n go here

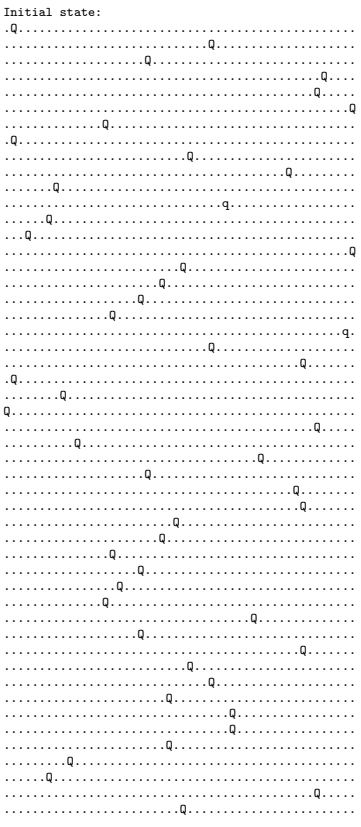
To include code in your report use this

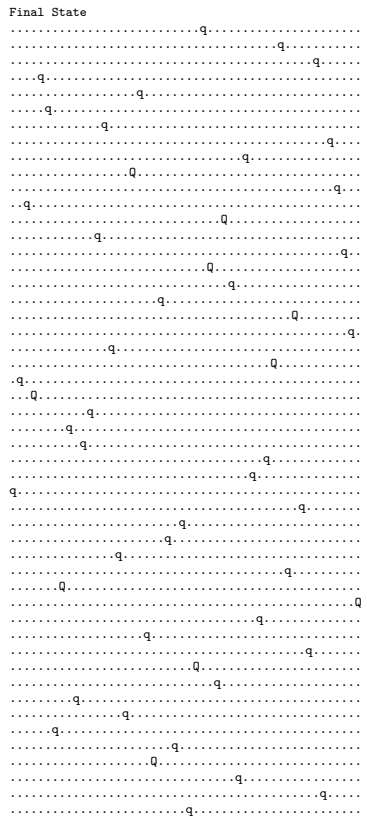
Copy code files into listings using the following commands (see .tex):

Exercise 1.1 output

```
Initial state:
......Q.
.....Q..
.....Q..
..Q.....
..Q.....
Q.......
..Q.....
..Q.....
```

```
Final State
.q......
...q....
.....q..
.......q
....Q...
Q.......
......Q.
..Q.....
```





Main.c

```
1 Your code here
```

SomeFile.c

```
1 Some other code here
```