Contents

[TSP 3](#_Toc59207142)

[Implementation Efficiency 3](#_Toc59207143)

[Peek into the results 3](#_Toc59207144)

[N-Queens 4](#_Toc59207145)

[Efficiency 4](#_Toc59207146)

## TSP

### Implementation Efficiency

The basic implementation was first profiled, and it was found that the largest amount of time was taken by two functions: *get\_distance()* and *is\_valid\_swap()*.

The first optimization we tried was to remove the math.sqrt from the distance calculation and use the squared distances itself. This doesn’t make any difference to the actual algorithm since the algorithm just makes decisions based on the best improvement, and whether we take the squre or square root, the best improvement would be the same edges. However, while this gave a significant improvement, for reasons cited below, we decided not to use this optimization.

The second optimization that we performed was to replace the power operator in python (\*\*) with the multiplication operator. We changed the distance calculation from :

(c2x - c1x) \*\*2 + (c2y - c1y) \*\*2

to

(c2x - c1x) \* (c2x - c1x) + (c2y - c1y) \* (c2y - c1y)

And the second version was much faster. This is probably because the operator \*\* calls into the generic pow() routine, which uses the Taylor series and table lookups to calculate the power. However, in this case, simple multiplication is likely to be much faster than Taylor series and table lookups.

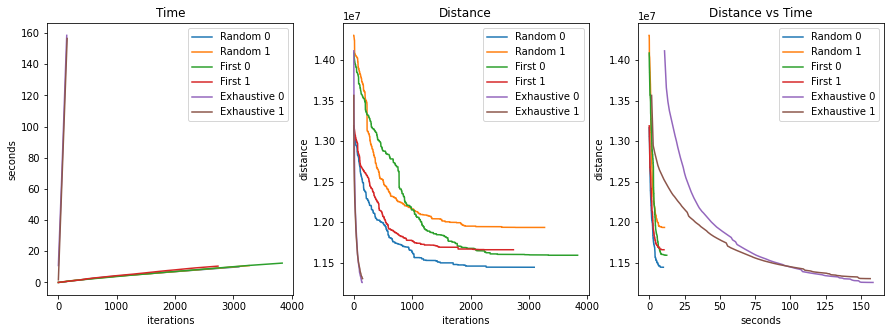
A LRU cache was used to cache the results of these functions, and in the case where there were no evictions, there was a significant improvement in the performance of these function, and each iteration became much faster. These are the results which demonstrate the effect of caching. The use of a cache makes each iteration about 60% faster.



We also compared the effect of using math.sqrt along with multiplication instead of the power operator, and we decided to use that (use math.sqrt along with multiplication), the effect of caching overshadowed any other improvements, and using math.sqrt meant that all the graphs would be in the correct scale.

### Peek into the results

This section provides a peek into the results to discuss the overall trend briefly. More detailed results will be presented later. In this section, we ran the three variants for 1 run, 2 restarts and 10,000 iterations. The number of cities was 819.



Here are the highlights:

1. The runs terminated much before the set limit of 10,000 iterations
2. Exhaustive search produced better results than the two variants of random methods, in a very few number of iterations
3. However, each iteration of exhaustive search was much more expensive, and overall, even with a greater number of iterations, the two variants completed under 20s, while exhaustive search took about 150 seconds
4. The completely random implementation showed more variation between the restarts than the first improvement variation
5. If better results are a necessity, exhaustive search may be the better option, however, for most applications, a quick-turn around may be more useful and the two variants can be used there.

## N-Queens

### Efficiency

#### Optimizations of cost calculation

With the N-Queens problem, the goal was to make the program as efficient as possible. The two bottleneck functions were *getHeuristicCostQueen* and *getHeuristicCost* which were taking up 79.10% and 19.59% of the time taken for the entire program.

Optimization of these functions was performed in three steps:

1. Avoiding use of math.fabs and using integer arithmetic resulted in 60% faster runs than the base implementation. This is because floating point operations are much slower than integer operations.
2. Using an LRU cache resulted in a 80% faster runs than the base installation.

Overall, the speedup achieved was about 5X.

In the table below getHeuristicCostQueen() was called 671104 times and GetHeuristicCost was called 12544 times.



Comparisons with LRU Cache have a non-linear growth as the number of operations grows, and may be difficult to relate to. Hence all comparisons have been done with both cached and non-cached versions.

#### Early Stopping Optimization

When sideways moves are not allowed the algorithm often gets stuck in a local minima earlier on, and cannot get out of it. It just keeps repeating the steps in the same neighborhood, but still continues executing till the maximum number of iterations without any hope of making any improvement. For example with 134 queens, 21 restarts, and a maximum of 100000 iterations, the algorithm didn’t find any improving moves after the following iteration in each restart:

116, 111, 127, 155, 114, 104, 117, 134, 141, 97, 123, 109, 118, 108, 96, 132, 100, 116, 101, 125, 241

On an average in the above example, after 123 iterations, no improving moves were found, but the algorithm still iterated through all the 100,000 iterations. As we can see, when sideways move was not allowed, it got stuck in local minimas very early. Continuing with the rest of the iterations was very wasteful.

Also, this meant that we had to try out different values for the number of iterations for the algorithm to stop in a meaningful interval of time.

All this could be avoided if we stopped early if we could figure out for certain that no further improving moves would be found. A version of early stopping was implemented as follows:

Algorithm for each iteration without early stopping is shown below.

Loop for *n* iterations:

*max\_cost* ß Highest number of conflicts for any queen  
*max\_candidate* ß All queens with *max\_cost*  
*candidate* ß Random choice from *max\_candidate*

*min\_cost* ß *max\_cost  
start\_min\_cost* ß *min\_cost*

for-each valid row position *pos\_i, candidate* can be moved to:  
 *state* ß resulting state if *candidate* was moved to *pos\_i*  
 *cost\_i* ß getHeuristicCostQueen(*pos\_i)* if *min\_cost* > *cost\_i*:  
 *min\_cost* ß *cost\_i*   
 *best\_pos* ß *[pos\_i]* elif *min\_cost* == *cost\_i* and *allow\_sideways*:  
 *best\_pos.append(pos\_i)*  
 elif *min\_cost == cost\_i and min\_cost < start\_min\_cost* and *!allow\_sideways:  
 best\_pos.append(pos\_i)*

if *best\_pos:*  
 // Some non worsening move has been found  
 *candidate\_solution[candidate] =* Random choice from *best\_pos  
 cost\_i* ß *getHeuristicCost(candidate\_solution)*  
else:  
 // No better solution found  
 *cost\_i* ß *getHeuristicCost(candidate\_solution)*

if *best\_cost* > *cost\_i*:  
 *best\_cost* = *cost\_i*

In the above algorithm, we select all the queens with *max\_cost* and store them in *max\_candidate*. If none of the candidates in *max\_candidate* yield an improving move, this restart will not be able to better the cost. This is because in the next iteration, *max\_candidate* will evaluate to exactly the same values, and since we’ve tried out all of them in previous iterations, and they haven’t yielded a better result, they will not yield a better result in this iteration as well.

This fact can be leveraged to implement an early exit. The early exit is implemented by modifying the algorithm as follows:

*queens\_tried\_set 🡨 SET()*  
Loop for *n* iterations:

*max\_cost* ß Highest number of conflicts for any queen  
*max\_candidate* ß All queens with *max\_cost*  
*candidate* ß Random choice from *max\_candidate  
queens\_tried\_set.add(candidate)*

*min\_cost* ß *max\_cost  
start\_min\_cost* ß *min\_cost*

for-each valid row position *pos\_i, candidate* can be moved to:  
 *state* ß resulting state if *candidate* was moved to *pos\_i*  
 *cost\_i* ß getHeuristicCostQueen(*pos\_i)* if *min\_cost* > *cost\_i*:  
 *min\_cost* ß *cost\_i*   
 *best\_pos* ß *[pos\_i]* elif *min\_cost* == *cost\_i* and *allow\_sideways*:  
 *best\_pos.append(pos\_i)*  
 elif *min\_cost == cost\_i and min\_cost < start\_min\_cost* and *!allow\_sideways:  
 best\_pos.append(pos\_i)*

if *best\_pos:*  
 // Some non worsening move has been found  
 *candidate\_solution[candidate] =* Random choice from *best\_pos  
 cost\_i* ß *getHeuristicCost(candidate\_solution)* *queens\_tried\_set.clear()* // Must clear this set because the state has changed  
 // We need to start afresh  
else:  
 // No better solution found  
 *cost\_i* ß *getHeuristicCost(candidate\_solution)*  if not *allow\_sideways* and *queens\_tried\_set == SET(max\_candidate)*:  
 // We’ve tried all moves possible, and the next iteration will just repeat this  
 // We can bail out early  
 break

if *best\_cost* > *cost\_i*:  
 *best\_cost* = *cost\_i*

In the above, we maintain a Set() to track which candidates we’ve already tried from max\_candidates. If we’ve tried all candidates from max\_candidates, and found no improving moves, then no better solution is possible in this restart. The next iteration will just pick one of the same candidates again and run through with it. If we detect that the set and max\_candidates are equal, we can bail out. If, however, an improving move is found, then the set needs to be cleared because the state has changed, and states that we had tried earlier that didn’t give us better results can now give better results in the altered board. The SET data structure was used because insertion and removal from a set are constant time operations, and comparison between two sets is O(n).

The benefits of this implementation were:

1. *Set a very high number of iterations by default, and we no longer have to experiment with different values for the number of iterations, because when improving moves are no longer possible, the algorithm will stop early.* However, we still wanted to compare the effect of setting the maximum number of iterations and vary that, so we didn’t use the aforementioned scheme.
2. Makes the program more efficient