Table of Contents

[Introduction 3](#_Toc59321506)

[TSP 3](#_Toc59321507)

[Implementation Efficiency 3](#_Toc59321508)

[Peek into the results 4](#_Toc59321509)

[N-Queens 5](#_Toc59321510)

[General Notes 5](#_Toc59321511)

[Efficiency 5](#_Toc59321512)

[Optimizations of cost calculation 5](#_Toc59321513)

[Early Stopping Optimization 6](#_Toc59321514)

[Removing the sideways steps 7](#_Toc59321515)

[Basic Statistics Summary and Discussion 8](#_Toc59321516)

[Sideways without caching 9](#_Toc59321517)

[No sideways without caching 10](#_Toc59321518)

[Sideways with caching 11](#_Toc59321519)

[No sideways with caching 11](#_Toc59321520)

[Run Time Distributions 11](#_Toc59321521)

[Sideways moves allowed without caching 11](#_Toc59321522)

[Sideways Not Allowed without Cache 13](#_Toc59321523)

[Sideways allowed with caching 15](#_Toc59321524)

[Sideways Not Allowed, cached 16](#_Toc59321525)

[Cached vs Non-Cached (sideways) 18](#_Toc59321526)

[Cached vs Non Cached (Sideways Not Allowed) 19](#_Toc59321527)

[Sideways vs Non-Sideways uncached 19](#_Toc59321530)

# Introduction

In this assignment, two problems were considered. A traveling salesman problem was implemented using local search and 2-opt. The supplied program for N-Queens was modified slightly and run-time diagrams plotted for N=54. To measure the time taken, calls to the python API *time.process\_time()* was inserted into the code. *process\_time()* reports the combination of User+Sys time, and not the wall time.

Local Search relies on being able perform many iterations, where each iteration runs very quickly. It was noticed during the evaluation that spending time to optimize each iteration so that it runs quickly has many benefits – we noticed a 16X speed up when we introduced certain optimizations. Practically that brought the run time down from several hours to several minutes.

Apart from optimizing iterations, a identifying a good early stopping criterion can help the algorithm a lot. In some cases, after a certain number of steps, the algorithm is guaranteed never to find better solutions, even if the iteration in the present runs are introduced. If we can identify such a condition, it would be prudent to terminate the iteration and restart a new one. We were able to find one such condition for the N-Queens where side-stepping was disabled, and that led to a significantly smaller number of iterations in each restart. Typically, what we found was that after a small multiple of N moves (where N is the number of queens, the multiple being a small integer), no improving moves could be found. Incorporating the early stopping criterion, we were able to avoid executing thousands of iterations when the number of iterations was set very high. ***More importantly, this also gives us a method to use arbitrarily high number of iterations, because we can be sure that if the algorithm gets stuck, it will identify it and bail out early, but if it keeps finding better solutions, the number of iterations can be pushed higher and higher.*** In this example of N-Queens, we were able to set the number of iterations to 100,000 and still arrive at the solution in a matter of hours, where this would have taken days if the early stopping criterion was not applied.

All experiments were done on a HP Z-Book with i7-6820HQ having 4 cores and 2 HyperThreads at 2.7GHZ, and 64 GB of RAM.

# 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.

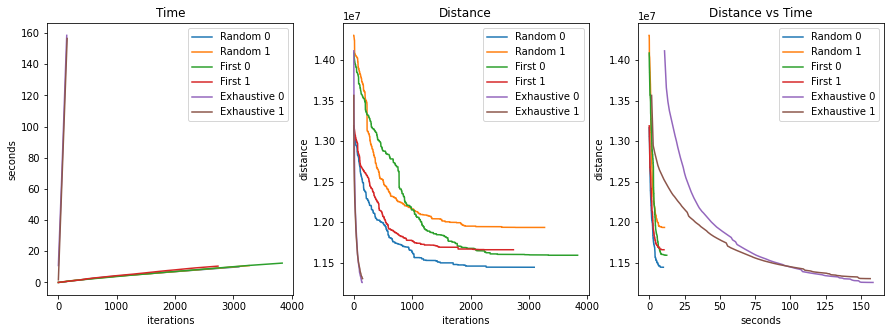
A LRU cache was used to cache the results of these functions, and in the case where there were no evictions, there was a significant improvement in the performance of these function, and each iteration became much faster. These are the results which demonstrate the effect of caching. The use of a cache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

## Results

# N-Queens

## General Notes

Run-length statistics and diagrams should be done in terms of number of operations where each operation has a fixed cost. For our purpose, we decided to calculate the number of times the function getHeuristicCostQueen() was called. getHeuristicCostQueen() however is not a constant time function, and varies with the number of queens. In our case, the number of queens was fixed at 54, and multiplied the number of calls to this function by 54 to arrive at a constant cost (since this is an O(n) operation and n=54).

The Jupyter notebook can be referred to for all data noted in this section. The notebook is cross-referenced.

We also implemented a version of getHeuristicCostQueen that involved caching results from previous calls. That gave us a big performance boost. Data has been presented from both the cached and non-cached versions.

## 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).

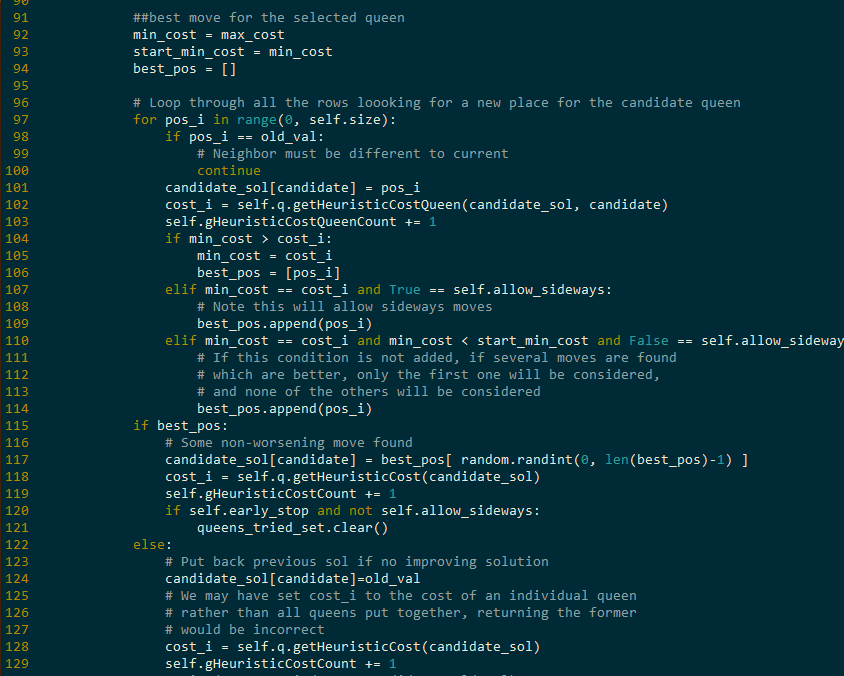
The benefits of this implementation were:

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

## Removing the sideways steps

Side-stepping was removed as per instructions. This had to be done carefully, and needed addition of another condition in line 110 in the snippet below. If this was not added, then if several moves of equal cost were found, all better than the current state, only the first one will be chosen always. This is not desired because this decreases the diversity we have artificially.

Another modification that had to be made was to return the actual cost of all conflicts in case we haven’t reached zero. In the snippet below, line 128 achieves the same.



## Basic Statistics Summary and Discussion

Basic statistics were collected for various number of iterations. Four combinations were tried:

1. Sideways with caching
2. Sideways without caching
3. No sideways with caching
4. No sideways without caching

For cached and non-cached algorithms, there is no change in the number of operations, and the only change is in the time taken. Hence the number of operations for cached items is not reproduced in this document, but is there in the jupyter notebook.

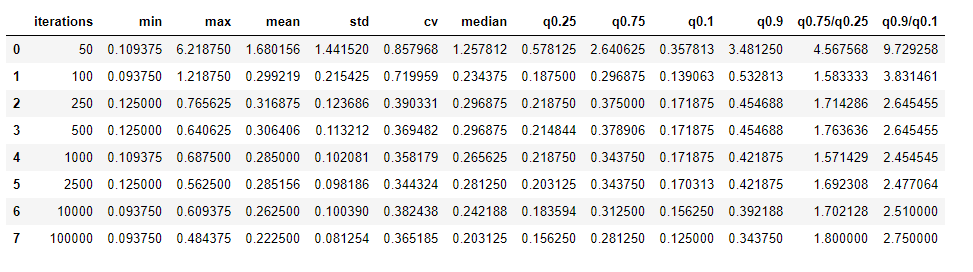
What we observe is that with 50 iterations, the success rate is low, it improves with 100 iterations, and achieves a peak somewhere between 100 and 250 iterations. We also see that with sideways moves, things are both much faster, and also produces better results. Beyond 250 iterations, having sideways moves achieved 100% success rate with few restarts, while not having sideways moves only produced successful results 94% of the time even with 100,000 iterations. The number of restarts required when sideways moves were not allowed were also higher (321 restarts on an average). By contrast, when sideways moves were allowed, no restarts were required with a reasonable iteration limit.

We also notice that caching significantly improved the run time, often by a factor of 2 or more. For example, when sideways moves are not enabled, when iterations is set at 100,000, the mean time taken without caching is 46.45 seconds, but with caching enabled, this was reduced to 29.0.

Allowing sideways moves was better not only in terms of success rate, but also was several orders of magnitude faster than non-sideways version. When caching was not enabled, non-sideways moves completed in 46.45 seconds, while the sideways version completed in 0.23 seconds. Even with caching enabled, the non-sideways version only completed in 29.0 seconds.

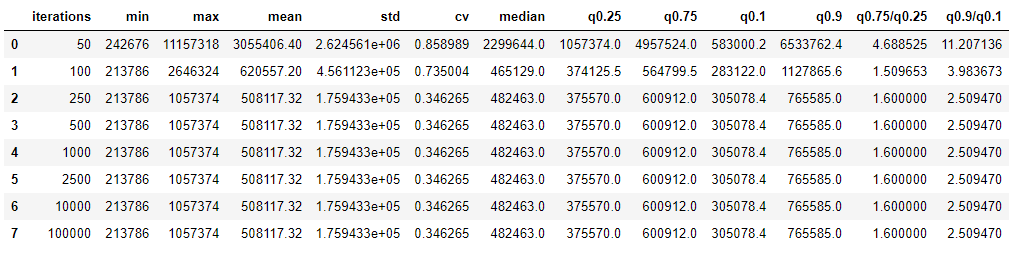
### Sideways without caching

#### Run Time



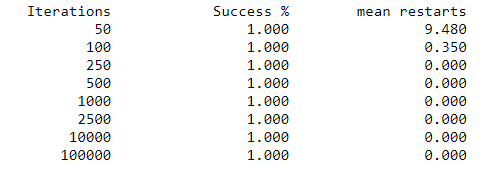
What we see above is that the standard deviation is very high for 50 iterations, is somewhat high for 100 iterations, and then falls low after that. This can be attributed to the fact that in most cases, 50 iterations didn’t produce results and needed a lot of restarts before finding the correct solution. Things improved with 100 iterations, but the ideal number was at 250 and beyond. The median run times also reflect a pattern that corroborates the same.

#### Number of operations



Here we can see the real difference, with 100 iterations, the number of operations is an order of magnitude less than the number of operations with 50 iterations.

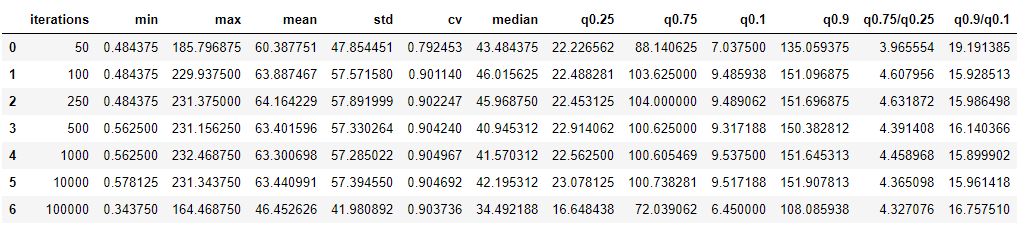
#### Success Rate



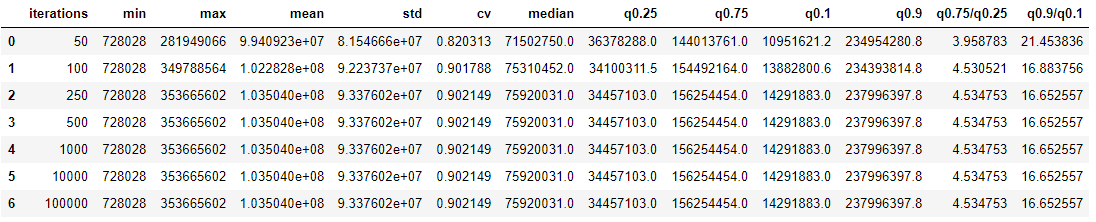
Here we can see that all runs were successful, however, with 50 iterations it needed quite a few restarts, and with 250 iterations and onwards, no restarts were required.

### No sideways without caching

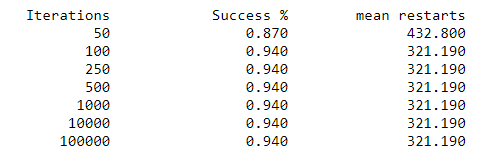
#### Run Time



#### Number of Operations



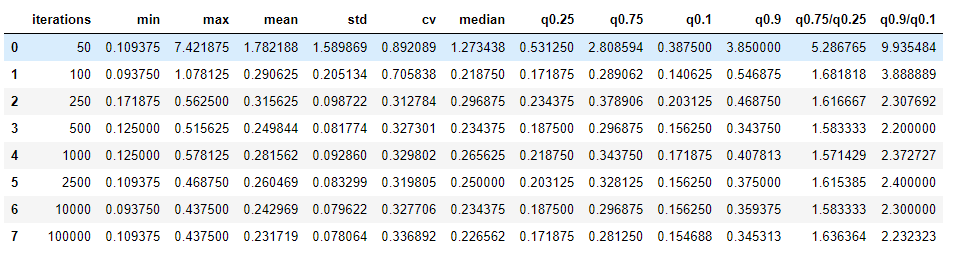
#### Success Rate



Overall, what we see here that the ideal number of iterations is 100+. Also what we see is that the success rate is lower than what sideways moves gave us. Also the running time is much higher than the sideways version – several orders of magnitude higher.

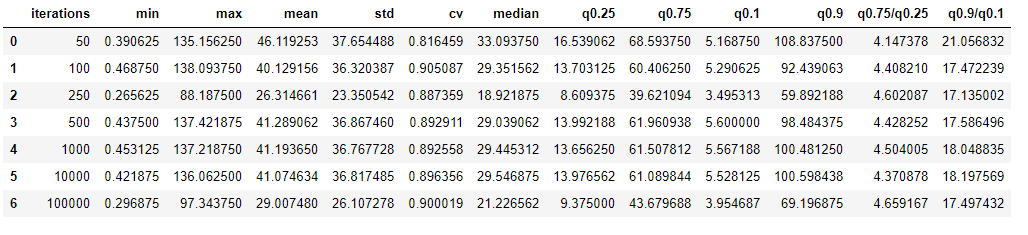
### Sideways with caching

#### Run Time



### No sideways with caching

#### Run Time



## Run Time Distributions

### Sideways moves allowed without caching

We observe that when iterations is set at 50, the performance is really bad and it takes a long time to reach a high probability of success. From earlier, we also saw that with 50, we never reach P(Solve)=100%. There is a marked increase when iterations is increased to 100, and then peaks out when iterations is set to 250.

Also, we see that iterations=50 performs several times more slowly than higher number of iterations.

We see some variation between the different iteration limits beyond 250 in the run-time distribution, but when the run-length distribution is observed, we see that the number of operations beyond 250 is always the same. We can attribute the difference in run time to randomness and other factors.

This also correlates with the 100% success rate in the first restart that we saw in the previous section.

#### Run Time Distribution [P(Solve)]

#### Run Time Distribution (Failure Rate)Run Length Distribution [P(Solve)]

#### Run Length Distribution (Failure Rate)

### Sideways Not Allowed without Cache

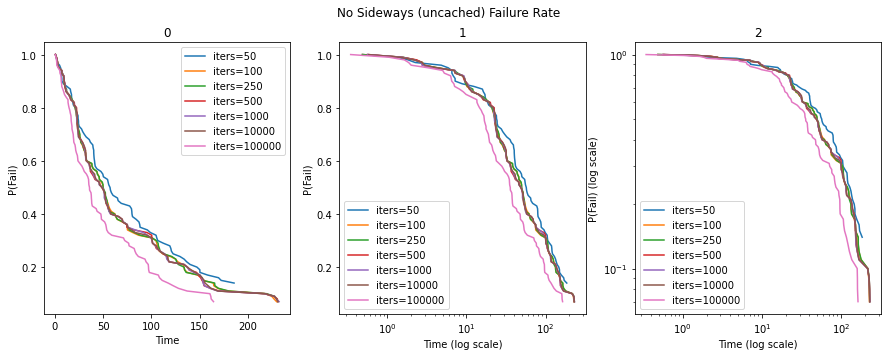
Here we see that when the number of iterations is set to 50, the algorithm converges slowly. Also, the algorithm is only able to reach P(Solve) of 87%, while for other iterations, we see that a P(Solve) of 94% is achieved. We also see that setting iterations to 100 performs significantly better than 50. The performance is peaks shortly after 100.

When we see the run-length diagram, we can see that the number of operations is almost exactly the same when the number of iterations is more than 100. The difference in time can boil down to random and other extraneous factors like machine load.

The other trend we observe for iterations=100 is that the distribution is identical with iterations > 100 till P(Solve) reaches 0.8, and after that it becomes increasingly difficult to get more solutions with iterations=100.

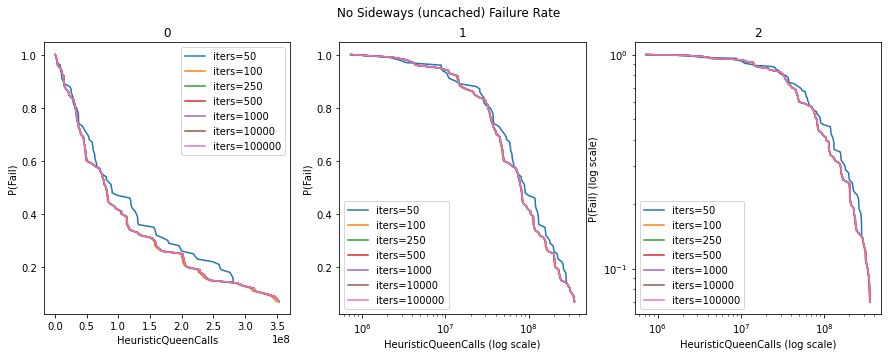
#### Run Time Diagram P(Solve)

#### Run Time Diagram (Failure Rate)



#### Run Length Diagram P(Solve)

#### Run Length Diagram (Failure Rate)



### Sideways allowed with caching

In this section, we only present the run time distributions because caching has no effect on the number of operations. The only difference is that the operations themselves can be cached instead of performed every time. This results in each iteration becoming faster, but the same number of iterations must be performed.

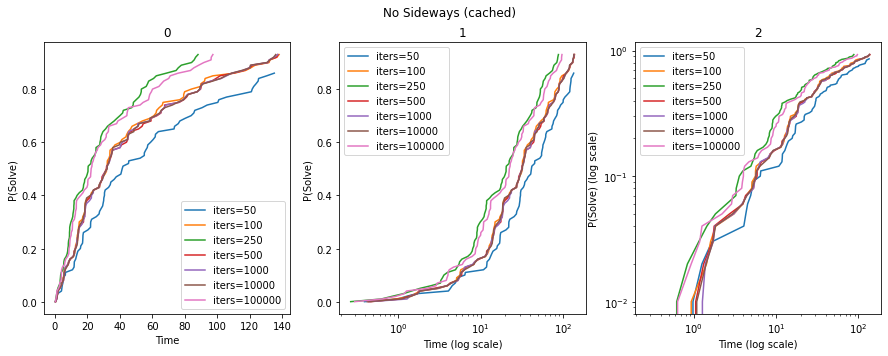
#### Run Time Distribution P(Solve)

#### Run Time Distribution (Failure Rate)

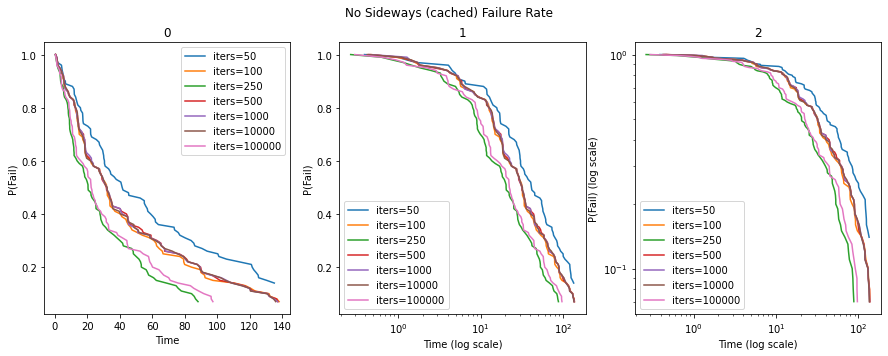
### Sideways Not Allowed, cached

We don’t go too much into details in this section. The overall trends are the same as discussed above. In the next section, we compare the diagrams side by side to see where the differences lie.

#### Run Time Distribution P(Solve)

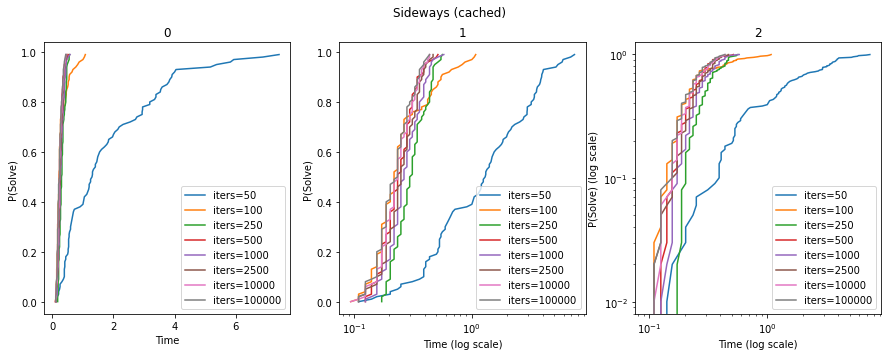


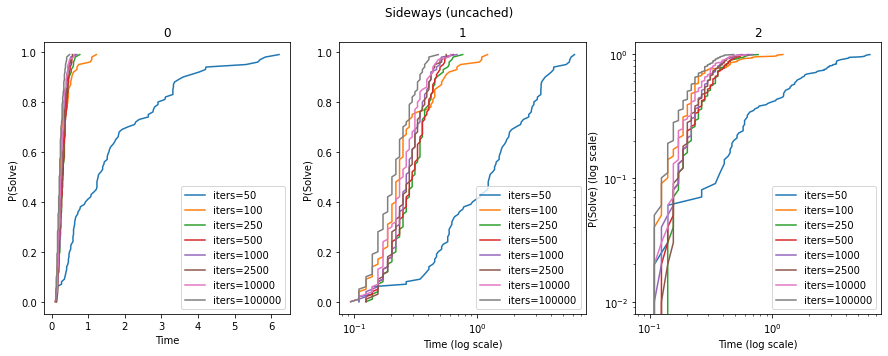
#### Run Time Distribution P(Fail)



### Cached vs Non-Cached (sideways)

Here we observe that there is no meaningful change in the run time between cached and non-cached versions when sideways moves are allowed. This is because when sideways moves are allowed, things are quite fast by themselves, and the other overheads are more than the benefits from caching.





### Cached vs Non Cached (Sideways Not Allowed)

Here we see that the cached version is significantly faster than the non-cached version of the same.

### 

### 

### Sideways vs Non-Sideways uncached

Here we see the following:

The version that allows sideways moves is significantly faster than the one that doesn’t allow side moves

The version that allows sideways moves also has a bettes probability of success for both high and low number of iterations

When sideways movement is allowed, there is a marked difference between iterations=50 and iterations=100. When sideways movement is not allowed, there is a difference between iterations=50 and iterations=100 in terms of runtime, but it is not as large. However, when it comes to P(Solve), when sideways moves are not allowed, there is a marked difference between iterations=50 and iterations=100

The last plot shows sideways vs non-sideways in the same graph, and we can see a marked difference in run time between the two. The graph is a little cluttered because of many iterations, but we see two distinct groups. A group that finishes very quickly - the sideways group, and the group that solves less quickly – the non-sideways group.

