Assignment 2: Local Search

Rajbir Bhattacharjee, R00195734

Table of Contents

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.

We first discuss the implementation and results of local search on TSP, and then we discuss the results on NQueens. Finally, we discuss the key take-aways and general patterns we noticed after the experiments.

TSP

3 variants of a local search to TSP were implemented. The first is a basic 2-opt version; throughout this report this version is referred to as the base version. The second variant is one where we choose an edge at random, and compare this with all other edges as candidates for replacement; through out the code this is referred to as variant1. The second variant is one where we choose an edge at random, and then iterate through all other edges, and perform the swap at the first combination we find that improves the current cost; this is referred to as variant 2.

For all of the above, 5 runs were executed, each run had 10 restarts and 10000 iterations.

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.

	restart_and_iterate	restart_and_iterate with cache	get_distance	math.sqrt removed	Replace ** with *	* with sqrt	get_distance with cache	is_valid_swap	is_valid_swap with cache
Calls	1	1		6004640			6004640	1498770	1498770
Run 1	12.39	5.35	6.12	5.4	3.77	4.68	0.07183	1.94	0.0447
Run 2	14.63	5.66	6.1	6.09	3.62	4.78	0.07997	1.5	0.04831
Run 3	14.03	5.69	6.04	5.98	3.59	4.74	0.08289	2.19	0.05213
Average	13.68333333	5.566666667	6.086666667	5.823333333	3.66	4.733333333	0.07823	1.876666667	0.04838
Improvement (%)		**59.31790499**		**4.326396495**	**39.86856517**	**22.23439211**	**98.71473165**		**97.42202487**

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.

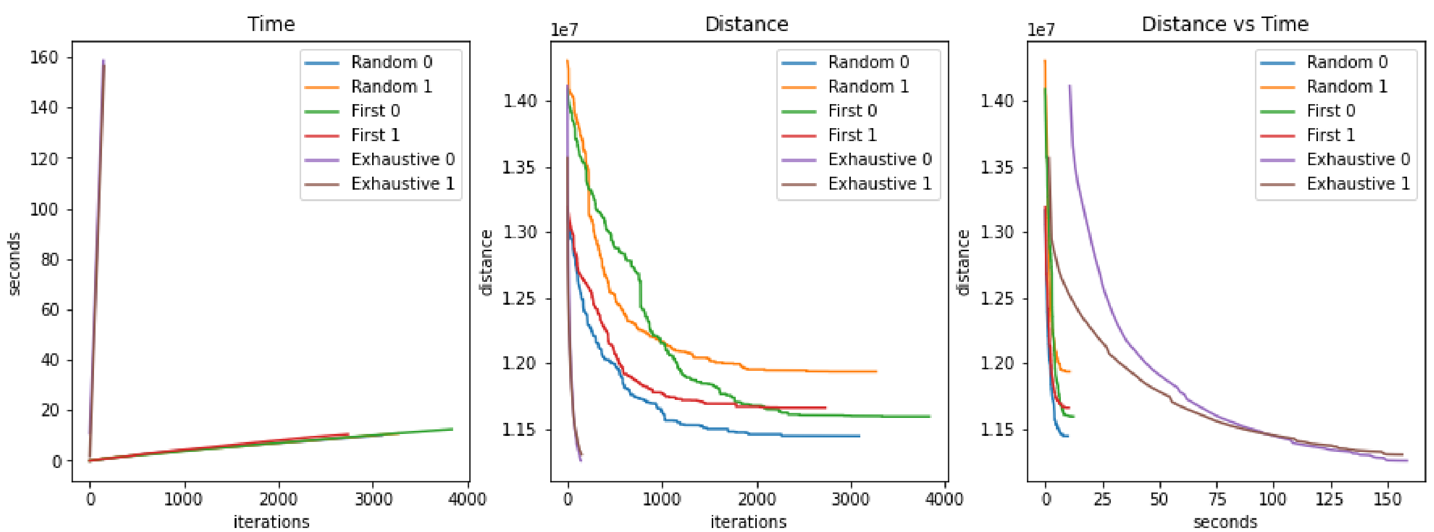


Figure 1Elapsed Processor Time vs Iterations

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.

The best distance from all the 5 runs is given by the following table:

instance	cities	algorithm	best-distance	mean-distance
inst-0	184	base	3452940.487764	3483211.185083
inst-0	184	variant1	3504535.839959	3560954.211323
inst-0	184	variant2	3564624.616259	3594153.936499
inst-13	352	base	6305151.506659	6324916.083904
inst-13	352	variant1	6364196.287549	6400748.188371
inst-13	352	variant2	6397159.378960	6441842.628152
inst-5	819	base	11091905.232430	11153406.098803
inst-5	819	variant1	11302910.062155	11350439.337075
inst-5	819	variant2	11331777.659586	11432677.888550

Results

Basic Statistics

Comparing the different variants in terms of best-distance

Here we see that the base variant produced the best results, followed by variant 1 and 2. What we also see is that variants 2 and 3 and more variation in the best distance produced. This is due to the fact that there is more randomness in the two other variants.

			best_distance							
			count	mean	std	min	25%	50%	75%	max
instance	cached	variant								
inst-0	False	base	5.0	3.483211e+06	27401.561743	3.452940e+06	3.461627e+06	3.486302e+06	3.493055e+06	3.522131e+06
		variant1	5.0	3.560954e+06	41682.079138	3.504536e+06	3.531718e+06	3.579031e+06	3.582463e+06	3.607023e+06
		variant2	5.0	3.594154e+06	19554.124732	3.564625e+06	3.588953e+06	3.594459e+06	3.607021e+06	3.615712e+06
	True	base	5.0	3.483211e+06	27401.561743	3.452940e+06	3.461627e+06	3.486302e+06	3.493055e+06	3.522131e+06
		variant1	5.0	3.560954e+06	41682.079138	3.504536e+06	3.531718e+06	3.579031e+06	3.582463e+06	3.607023e+06
		variant2	5.0	3.594154e+06	19554.124732	3.564625e+06	3.588953e+06	3.594459e+06	3.607021e+06	3.615712e+06
inst-13	False	base	5.0	6.324916e+06	28343.936094	6.305152e+06	6.309133e+06	6.312294e+06	6.323959e+06	6.374044e+06
		variant1	5.0	6.400748e+06	27786.048108	6.364196e+06	6.388281e+06	6.394371e+06	6.425151e+06	6.431742e+06
		variant2	5.0	6.441843e+06	49424.881214	6.397159e+06	6.417805e+06	6.425238e+06	6.444003e+06	6.525008e+06
	True	base	5.0	6.324916e+06	28343.936094	6.305152e+06	6.309133e+06	6.312294e+06	6.323959e+06	6.374044e+06
		variant1	5.0	6.400748e+06	27786.048108	6.364196e+06	6.388281e+06	6.394371e+06	6.425151e+06	6.431742e+06
		variant2	5.0	6.441843e+06	49424.881214	6.397159e+06	6.417805e+06	6.425238e+06	6.444003e+06	6.525008e+06
inst-5	False	base	5.0	1.115341e+07	36307.619408	1.109191e+07	1.115708e+07	1.116426e+07	1.116558e+07	1.118821e+07
		variant1	5.0	1.135044e+07	54096.755444	1.130291e+07	1.131195e+07	1.132049e+07	1.139532e+07	1.142153e+07
		variant2	5.0	1.143268e+07	74494.355171	1.133178e+07	1.140755e+07	1.142711e+07	1.146176e+07	1.153519e+07
	True	base	5.0	1.115341e+07	36307.619408	1.109191e+07	1.115708e+07	1.116426e+07	1.116558e+07	1.118821e+07
		variant1	5.0	1.135044e+07	54096.755444	1.130291e+07	1.131195e+07	1.132049e+07	1.139532e+07	1.142153e+07
		variant2	5.0	1.143268e+07	74494.355171	1.133178e+07	1.140755e+07	1.142711e+07	1.146176e+07	1.153519e+07

Figure 2Chart to illustrate how different variants compare when it comes to solution quality

Comparing the effects of caching on the different variants

Here we see that for the base variant, caching improved the performance significantly (between 20 and 30%), however, for variants 2 and 3, caching didn't have a big impact and sometimes actually degraded performance.

			process_time							
			count	mean	std	min	25%	50%	75%	max
instance	variant	cached								
inst-0	base	False	5.0	31.896875	2.652516	29.375000	30.828125	30.968750	31.968750	36.343750
		True	5.0	22.584375	0.607228	21.671875	22.500000	22.625000	22.765625	23.359375
	variant1	False	5.0	9.818750	0.454013	9.093750	9.671875	10.031250	10.062500	10.234375
		True	5.0	9.728125	0.956104	8.359375	9.453125	9.812500	10.015625	11.000000
	variant2	False	5.0	10.412500	0.270543	10.078125	10.203125	10.484375	10.546875	10.750000
		True	5.0	9.996875	0.564925	9.468750	9.656250	9.781250	10.187500	10.890625
inst-13	base	False	5.0	205.350000	11.371273	194.328125	195.218750	202.734375	215.859375	218.609375
		True	5.0	157.190625	15.452894	140.203125	151.125000	154.703125	157.765625	182.156250
	variant1	False	5.0	32.237500	3.301156	29.187500	30.109375	30.734375	33.968750	37.187500
		True	5.0	34.656250	3.655849	30.671875	30.921875	35.687500	37.671875	38.328125
	variant2	False	5.0	33.906250	1.252244	32.828125	33.296875	33.515625	33.843750	36.046875
		True	5.0	35.634375	3.590821	32.062500	32.656250	34.906250	38.062500	40.484375
inst-5	base	False	5.0	2089.825000	61.155074	2010.734375	2058.875000	2091.343750	2113.625000	2174.546875
		True	5.0	1588.146875	144.123185	1449.703125	1517.968750	1560.187500	1583.500000	1829.375000
	variant1	False	5.0	159.081250	6.889884	148.125000	159.578125	160.109375	160.328125	167.265625
		True	5.0	231.759375	118.888956	154.671875	176.187500	192.031250	193.312500	442.593750
	variant2	False	5.0	164.768750	10.036672	153.875000	155.390625	168.218750	168.546875	177.812500
		True	5.0	242.734375	109.046306	161.468750	205.000000	205.609375	206.843750	434.750000

Figure 3Chart to illustrate how caching affects the running time

Comparing run times with the different variants

Here we see that variant1 and variant2 were an order of magnitude faster than base. Variant1 was marginally vaster than variant2.

			process_time							
			count	mean	std	min	25%	50%	75%	max
instance	cached	variant								
inst-0	**False**	**base**	5.0	31.896875	2.652516	29.375000	30.828125	30.968750	31.968750	36.343750
		variant1	5.0	9.818750	0.454013	9.093750	9.671875	10.031250	10.062500	10.234375
		variant2	5.0	10.412500	0.270543	10.078125	10.203125	10.484375	10.546875	10.750000
	True	**base**	5.0	22.584375	0.607228	21.671875	22.500000	22.625000	22.765625	23.359375
		variant1	5.0	9.728125	0.956104	8.359375	9.453125	9.812500	10.015625	11.000000
		variant2	5.0	9.996875	0.564925	9.468750	9.656250	9.781250	10.187500	10.890625
inst-13	**False**	**base**	5.0	205.350000	11.371273	194.328125	195.218750	202.734375	215.859375	218.609375
		variant1	5.0	32.237500	3.301156	29.187500	30.109375	30.734375	33.968750	37.187500
		variant2	5.0	33.906250	1.252244	32.828125	33.296875	33.515625	33.843750	36.046875
	True	**base**	5.0	157.190625	15.452894	140.203125	151.125000	154.703125	157.765625	182.156250
		variant1	5.0	34.656250	3.655849	30.671875	30.921875	35.687500	37.671875	38.328125
		variant2	5.0	35.634375	3.590821	32.062500	32.656250	34.906250	38.062500	40.484375
inst-5	**False**	**base**	5.0	2089.825000	61.155074	2010.734375	2058.875000	2091.343750	2113.625000	2174.546875
		variant1	5.0	159.081250	6.889884	148.125000	159.578125	160.109375	160.328125	167.265625
		variant2	5.0	164.768750	10.036672	153.875000	155.390625	168.218750	168.546875	177.812500
	True	**base**	5.0	1588.146875	144.123185	1449.703125	1517.968750	1560.187500	1583.500000	1829.375000
		variant1	5.0	231.759375	118.888956	154.671875	176.187500	192.031250	193.312500	442.593750
		variant2	5.0	242.734375	109.046306	161.468750	205.000000	205.609375	206.843750	434.750000

Figure 4Chart to illustrate how different variants compare when it comes to running time

Comparing mean run times

For all the 5 runs, the mean run-time was compared with the different variants, with both cached and non-cached versions. What we see here is that variants 1 and 2 performed significantly better than the base 2-opt implementation. Where we see a difference is in the run times of cached and non-cached versions. Here we can see that for the basic implementation of 2-opt, the cached version was significantly faster than the non-cached version. However, for the two other variants, the cached version actually ran slower than the non-cached version. This may be because in these two, the number of operations is significantly reduced, and the costs associated with maintaining the cache do not justify the benefits of the time saved. However, this was still worth a try because implementing caching in python is just a single line of code and is easy to try out.

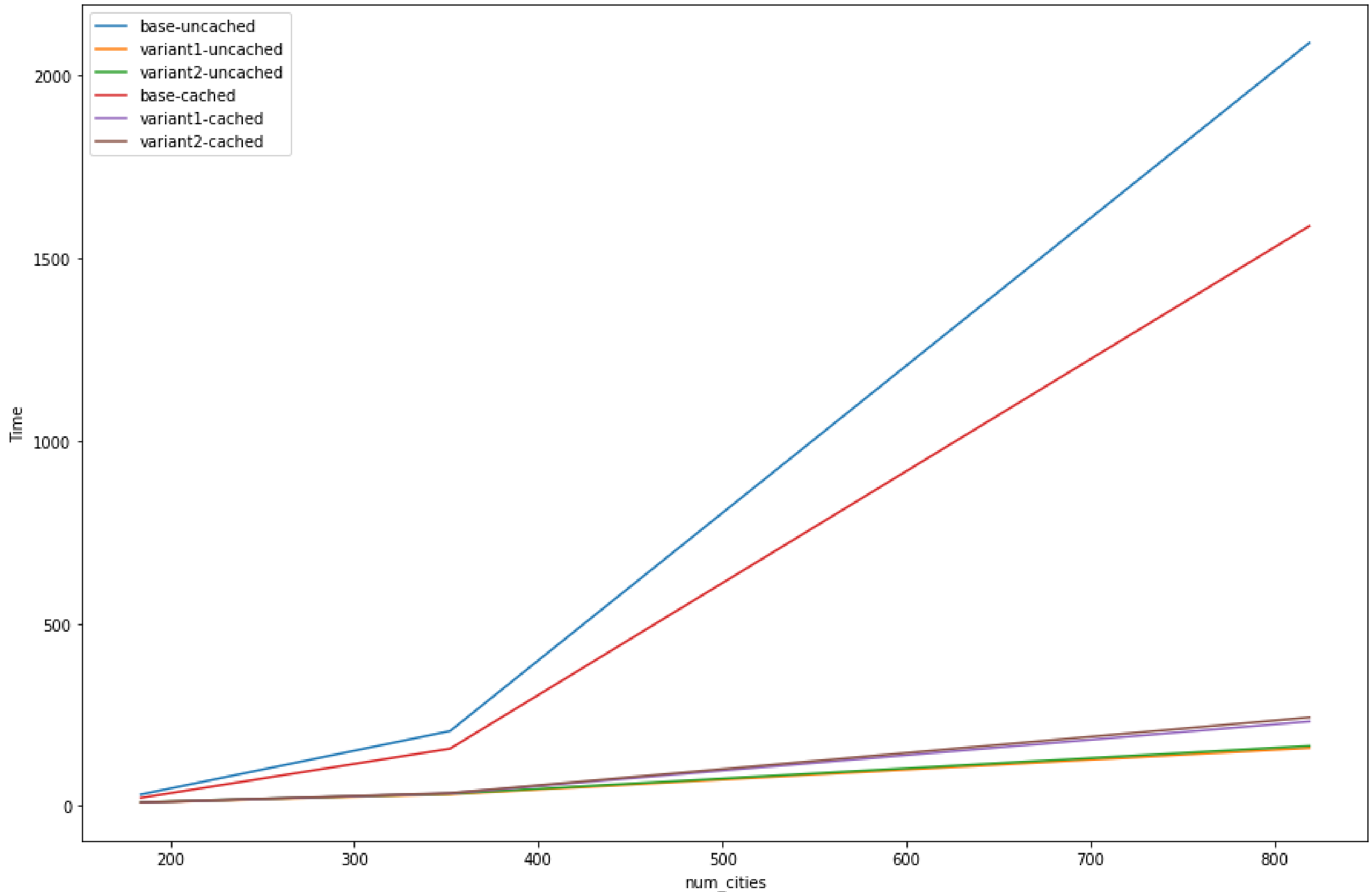


Figure 5Mean Run Time vs Number of Cities

Detailed Run Time Plots

The first thing we observe in the ‘best-distance vs iterations’ plot is that the base algorithm is able to produce better results than variants 1 and 2, and it does so in a lesser number of iterations. However, the fact that in lesser number of iterations is misleading because the total time taken by the base algorithm is more than the two variants. This is due to the fact that each iteration of the base implantation takes more time. This becomes clear in the ‘best-distance vs time’ graph, and even clearer in the ‘iterations vs run-time’ graph. Variants 1 and 2 have similar performances. The difference in performance becomes more pronounced as the number of cities increases. Only the detailed plots of the non-cached versions are discussed below because the greatest difference must be visible in the non-cached versions.

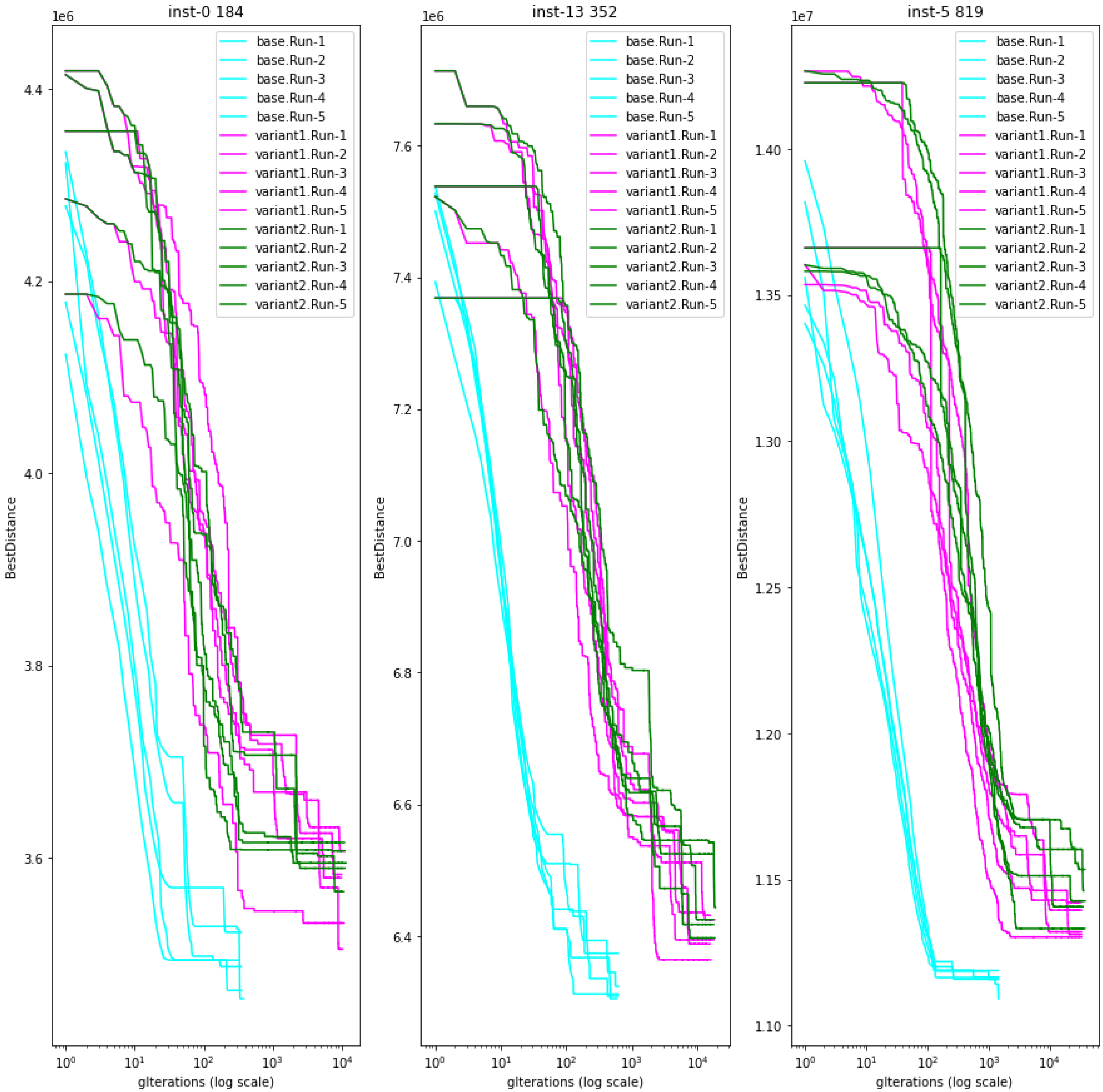


Figure 6Best Distance Achieved vs Iteration (all restarts cumulated)

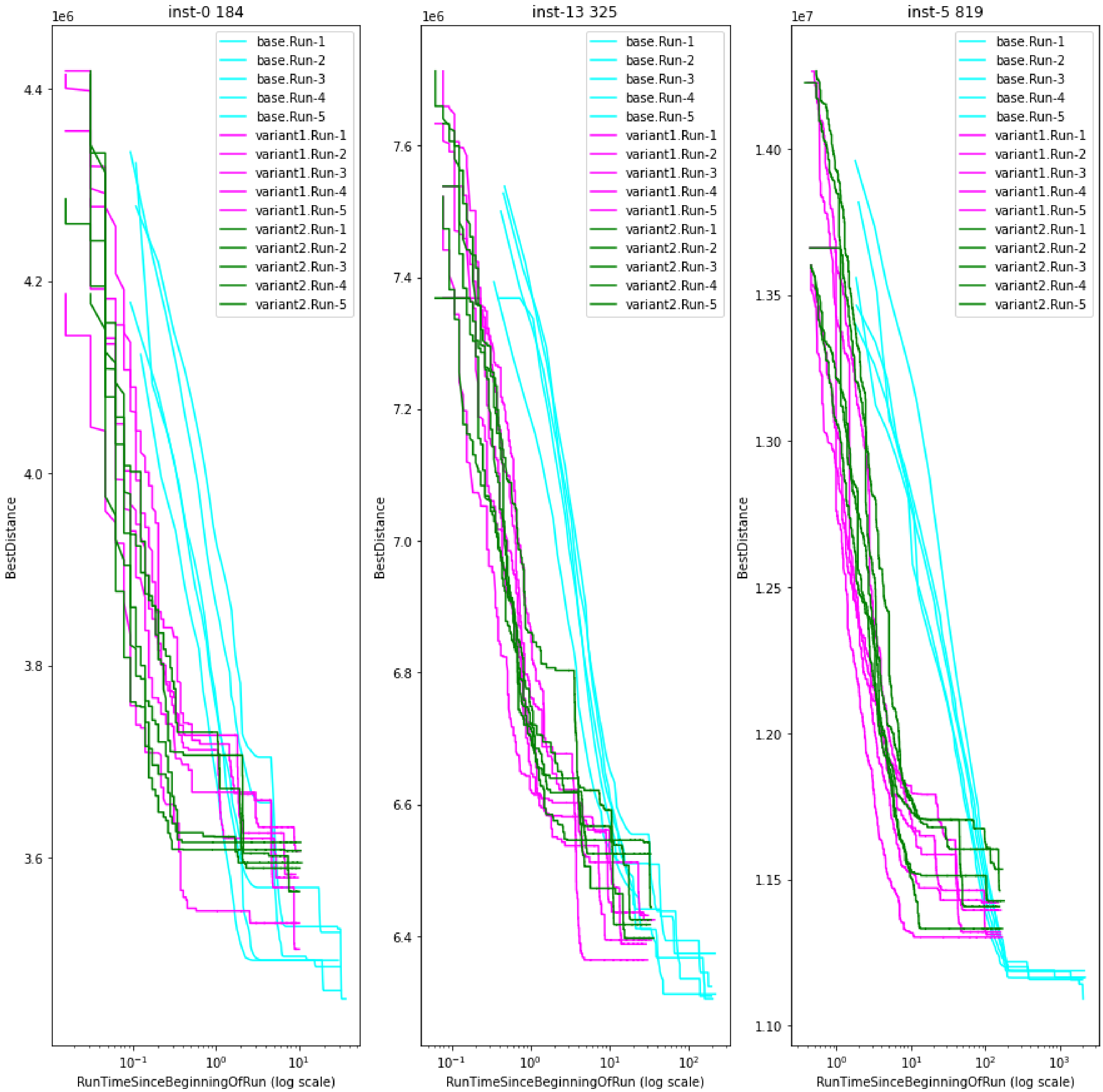


Figure 7Best distance vs Process Time since beginning of run (all restarts cumulated)

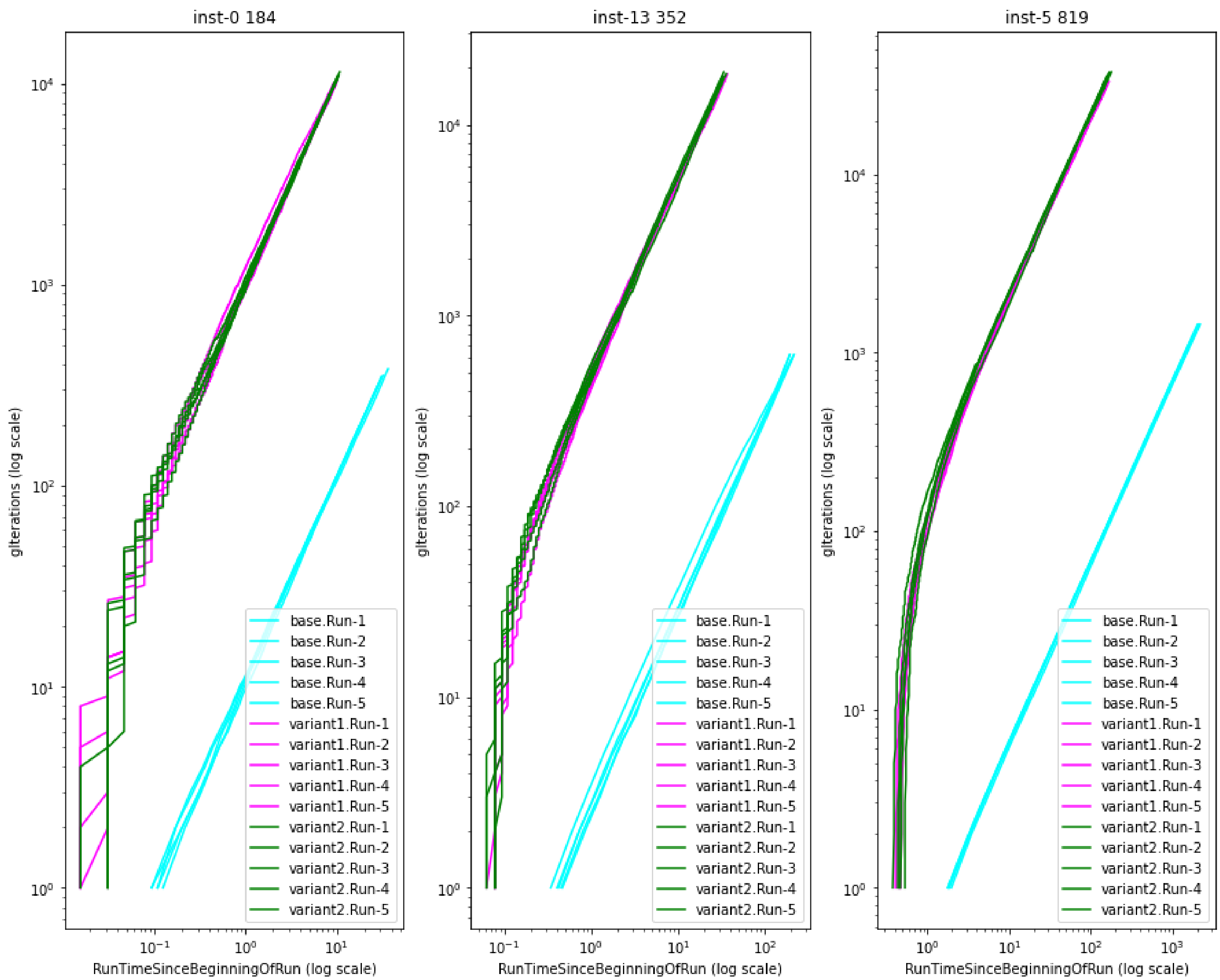


Figure 8Chart to illustrate iteration speed, cumulated number of iterations vs elapsed Processor time

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.

GetHeuristicCostQueen	**671104 calls**	**Base**	**use abs() instead of fabs()**	**Use Integers Comparisons**	**LRU Cache**
	Run 1	33.79	23.27	11.63	7.32
	Run 2	31.96	23.25	11.85	7.35
	Run 3	30.86	23.26	11.54	7.32
	Average	32.20333333	23.26	11.67333333	7.33
	Improvement from Base (%)		**27.77145223**	**63.75116448**	**77.23838112**
getHeuristicCost	**12544 calls**	**Base**	**use abs() instead of fabs()**	**Use Integers Comparisons**	**LRU Cache**
	Run 1	8.32	5.69	3.02	1.31
	Run 2	7.92	5.67	3.08	1.31
	Run 3	7.69	5.69	3	1.37
	Average	7.976666667	5.683333333	3.033333333	1.33
	Improvement from Base (%)		**28.75052236**	**61.97241956**	**83.32636858**

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 ← Highest number of conflicts for any queen
max_candidate ← All queens with max_cost

candidate ← Random choice from max_candidate

min_cost ← max_cost
start_min_cost ← min_cost

for-each valid row position pos_i, candidate can be moved to:
        state ← resulting state if candidate was moved to pos_i
        cost_i ← getHeuristicCostQueen(pos_i)
        if min_cost > cost_i:
                min_cost ←   cost_i
                best_pos ← [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 ←   getHeuristicCost(candidate_solution)
else:
        // No better solution found
        cost_i ←   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 ← Highest number of conflicts for any queen
    max_candidate ← All queens with max_cost

    candidate ← Random choice from max_candidate
    queens_tried_set.add(candidate)

    min_cost ← max_cost
    start_min_cost ← min_cost

    for-each valid row position pos_i, candidate can be moved to:
            state ← resulting state if candidate was moved to pos_i
            cost_i ← getHeuristicCostQueen(pos_i)
            if min_cost > cost_i:
                    min_cost ←   cost_i
                    best_pos ← [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 ←   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 ←   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.

```
        ##best move for the selected queen
        min_cost = max_cost
        start_min_cost = min_cost
        best_pos = []

        # Loop through all the rows loooking for a new place for the candidate queen
        for pos_i in range(0, self.size):
            if pos_i == old_val:
                # Neighbor must be different to current
                continue
            candidate_sol[candidate] = pos_i
            cost_i = self.q.getHeuristicCostQueen(candidate_sol, candidate)
            self.gHeuristicCostQueenCount += 1
            if min_cost > cost_i:
                min_cost = cost_i
                best_pos = [pos_i]
            elif min_cost == cost_i and True == self.allow_sideways:
                # Note this will allow sideways moves
                best_pos.append(pos_i)
            elif min_cost == cost_i and min_cost < start_min_cost and False == self.allow_sideway
                # If this condition is not added, if several moves are found
                # which are better, only the first one will be considered,
                # and none of the others will be considered
                best_pos.append(pos_i)
        if best_pos:
            # Some non-worsening move found
            candidate_sol[candidate] = best_pos[ random.randint(0, len(best_pos)-1) ]
            cost_i = self.q.getHeuristicCost(candidate_sol)
            self.gHeuristicCostCount += 1
            if self.early_stop and not self.allow_sideways:
                queens_tried_set.clear()
        else:
            # Put back previous sol if no improving solution
            candidate_sol[candidate]=old_val
            # We may have set cost_i to the cost of an individual queen
            # rather than all queens put together, returning the former
            # would be incorrect
            cost_i = self.q.getHeuristicCost(candidate_sol)
            self.gHeuristicCostCount += 1
```

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

	iterations	min	max	mean	std	cv	median	q0.25	q0.75	q0.1	q0.9	q0.75/q0.25	q0.9/q0.1
0	50	0.109375	6.218750	1.680156	1.441520	0.857968	1.257812	0.578125	2.640625	0.357813	3.481250	4.567568	9.729258
1	100	0.093750	1.218750	0.299219	0.215425	0.719959	0.234375	0.187500	0.296875	0.139063	0.532813	1.583333	3.831461
2	250	0.125000	0.765625	0.316875	0.123686	0.390331	0.296875	0.218750	0.375000	0.171875	0.454688	1.714286	2.645455
3	500	0.125000	0.640625	0.306406	0.113212	0.369482	0.296875	0.214844	0.378906	0.171875	0.454688	1.763636	2.645455
4	1000	0.109375	0.687500	0.285000	0.102081	0.358179	0.265625	0.218750	0.343750	0.171875	0.421875	1.571429	2.454545
5	2500	0.125000	0.562500	0.285156	0.098186	0.344324	0.281250	0.203125	0.343750	0.170313	0.421875	1.692308	2.477064
6	10000	0.093750	0.609375	0.262500	0.100390	0.382438	0.242188	0.183594	0.312500	0.156250	0.392188	1.702128	2.510000
7	100000	0.093750	0.484375	0.222500	0.081254	0.365185	0.203125	0.156250	0.281250	0.125000	0.343750	1.800000	2.750000

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

	iterations	min	max	mean	std	cv	median	q0.25	q0.75	q0.1	q0.9	q0.75/q0.25	q0.9/q0.1
0	50	242676	11157318	3055406.40	2.624561e+06	0.858989	2299644.0	1057374.0	4957524.0	583000.2	6533762.4	4.688525	11.207136
1	100	213786	2646324	620557.20	4.561123e+05	0.735004	465129.0	374125.5	564799.5	283122.0	1127865.6	1.509653	3.983673
2	250	213786	1057374	508117.32	1.759433e+05	0.346265	482463.0	375570.0	600912.0	305078.4	765585.0	1.600000	2.509470
3	500	213786	1057374	508117.32	1.759433e+05	0.346265	482463.0	375570.0	600912.0	305078.4	765585.0	1.600000	2.509470
4	1000	213786	1057374	508117.32	1.759433e+05	0.346265	482463.0	375570.0	600912.0	305078.4	765585.0	1.600000	2.509470
5	2500	213786	1057374	508117.32	1.759433e+05	0.346265	482463.0	375570.0	600912.0	305078.4	765585.0	1.600000	2.509470
6	10000	213786	1057374	508117.32	1.759433e+05	0.346265	482463.0	375570.0	600912.0	305078.4	765585.0	1.600000	2.509470
7	100000	213786	1057374	508117.32	1.759433e+05	0.346265	482463.0	375570.0	600912.0	305078.4	765585.0	1.600000	2.509470

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

Iterations	Success %	mean restarts
50	1.000	9.480
100	1.000	0.350
250	1.000	0.000
500	1.000	0.000
1000	1.000	0.000
2500	1.000	0.000
10000	1.000	0.000
100000	1.000	0.000

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

	iterations	min	max	mean	std	cv	median	q0.25	q0.75	q0.1	q0.9	q0.75/q0.25	q0.9/q0.1
0	50	0.484375	185.796875	60.387751	47.854451	0.792453	43.484375	22.226562	88.140625	7.037500	135.059375	3.965554	19.191385
1	100	0.484375	229.937500	63.887467	57.571580	0.901140	46.015625	22.488281	103.625000	9.485938	151.096875	4.607956	15.928513
2	250	0.484375	231.375000	64.164229	57.891999	0.902247	45.968750	22.453125	104.000000	9.489062	151.696875	4.631872	15.986498
3	500	0.562500	231.156250	63.401596	57.330264	0.904240	40.945312	22.914062	100.625000	9.317188	150.382812	4.391408	16.140366
4	1000	0.562500	232.468750	63.300698	57.285022	0.904967	41.570312	22.562500	100.605469	9.537500	151.645313	4.458968	15.899902
5	10000	0.578125	231.343750	63.440991	57.394550	0.904692	42.195312	23.078125	100.738281	9.517188	151.907813	4.365098	15.961418
6	100000	0.343750	164.468750	46.452626	41.980892	0.903736	34.492188	16.648438	72.039062	6.450000	108.085938	4.327076	16.757510

Number of Operations

	iterations	min	max	mean	std	cv	median	q0.25	q0.75	q0.1	q0.9	q0.75/q0.25	q0.9/q0.1
0	50	728028	281949066	9.940923e+07	8.154666e+07	0.820313	71502750.0	36378288.0	144013761.0	10951621.2	234954280.8	3.958783	21.453836
1	100	728028	349788564	1.022828e+08	9.223737e+07	0.901788	75310452.0	34100311.5	154492164.0	13882800.6	234393814.8	4.530521	16.883756
2	250	728028	353665602	1.035040e+08	9.337602e+07	0.902149	75920031.0	34457103.0	156254454.0	14291883.0	237996397.8	4.534753	16.652557
3	500	728028	353665602	1.035040e+08	9.337602e+07	0.902149	75920031.0	34457103.0	156254454.0	14291883.0	237996397.8	4.534753	16.652557
4	1000	728028	353665602	1.035040e+08	9.337602e+07	0.902149	75920031.0	34457103.0	156254454.0	14291883.0	237996397.8	4.534753	16.652557
5	10000	728028	353665602	1.035040e+08	9.337602e+07	0.902149	75920031.0	34457103.0	156254454.0	14291883.0	237996397.8	4.534753	16.652557
6	100000	728028	353665602	1.035040e+08	9.337602e+07	0.902149	75920031.0	34457103.0	156254454.0	14291883.0	237996397.8	4.534753	16.652557

Success Rate

Iterations	Success %	mean restarts
50	0.870	432.800
100	0.940	321.190
250	0.940	321.190
500	0.940	321.190
1000	0.940	321.190
10000	0.940	321.190
100000	0.940	321.190

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

	iterations	min	max	mean	std	cv	median	q0.25	q0.75	q0.1	q0.9	q0.75/q0.25	q0.9/q0.1
0	50	0.109375	7.421875	1.782188	1.589869	0.892089	1.273438	0.531250	2.808594	0.387500	3.850000	5.286765	9.935484
1	100	0.093750	1.078125	0.290625	0.205134	0.705838	0.218750	0.171875	0.289062	0.140625	0.546875	1.681818	3.888889
2	250	0.171875	0.562500	0.315625	0.098722	0.312784	0.296875	0.234375	0.378906	0.203125	0.468750	1.616667	2.307692
3	500	0.125000	0.515625	0.249844	0.081774	0.327301	0.234375	0.187500	0.296875	0.156250	0.343750	1.583333	2.200000
4	1000	0.125000	0.578125	0.281562	0.092860	0.329802	0.265625	0.218750	0.343750	0.171875	0.407813	1.571429	2.372727
5	2500	0.109375	0.468750	0.260469	0.083299	0.319805	0.250000	0.203125	0.328125	0.156250	0.375000	1.615385	2.400000
6	10000	0.093750	0.437500	0.242969	0.079622	0.327706	0.234375	0.187500	0.296875	0.156250	0.359375	1.583333	2.300000
7	100000	0.109375	0.437500	0.231719	0.078064	0.336892	0.226562	0.171875	0.281250	0.154688	0.345313	1.636364	2.232323

No sideways with caching

Run Time

	iterations	min	max	mean	std	cv	median	q0.25	q0.75	q0.1	q0.9	q0.75/q0.25	q0.9/q0.1
0	50	0.390625	135.156250	46.119253	37.654488	0.816459	33.093750	16.539062	68.593750	5.168750	108.837500	4.147378	21.056832
1	100	0.468750	138.093750	40.129156	36.320387	0.905087	29.351562	13.703125	60.406250	5.290625	92.439063	4.408210	17.472239
2	250	0.265625	88.187500	26.314661	23.350542	0.887359	18.921875	8.609375	39.621094	3.495313	59.892188	4.602087	17.135002
3	500	0.437500	137.421875	41.289062	36.867460	0.892911	29.039062	13.992188	61.960938	5.600000	98.484375	4.428252	17.586496
4	1000	0.453125	137.218750	41.193650	36.767728	0.892558	29.445312	13.656250	61.507812	5.567188	100.481250	4.504005	18.048835
5	10000	0.421875	136.062500	41.074634	36.817485	0.896356	29.546875	13.976562	61.089844	5.528125	100.598438	4.370878	18.197569
6	100000	0.296875	97.343750	29.007480	26.107278	0.900019	21.226562	9.375000	43.679688	3.954687	69.196875	4.659167	17.497432

Run Time Distributions

Sideways moves allowed without caching

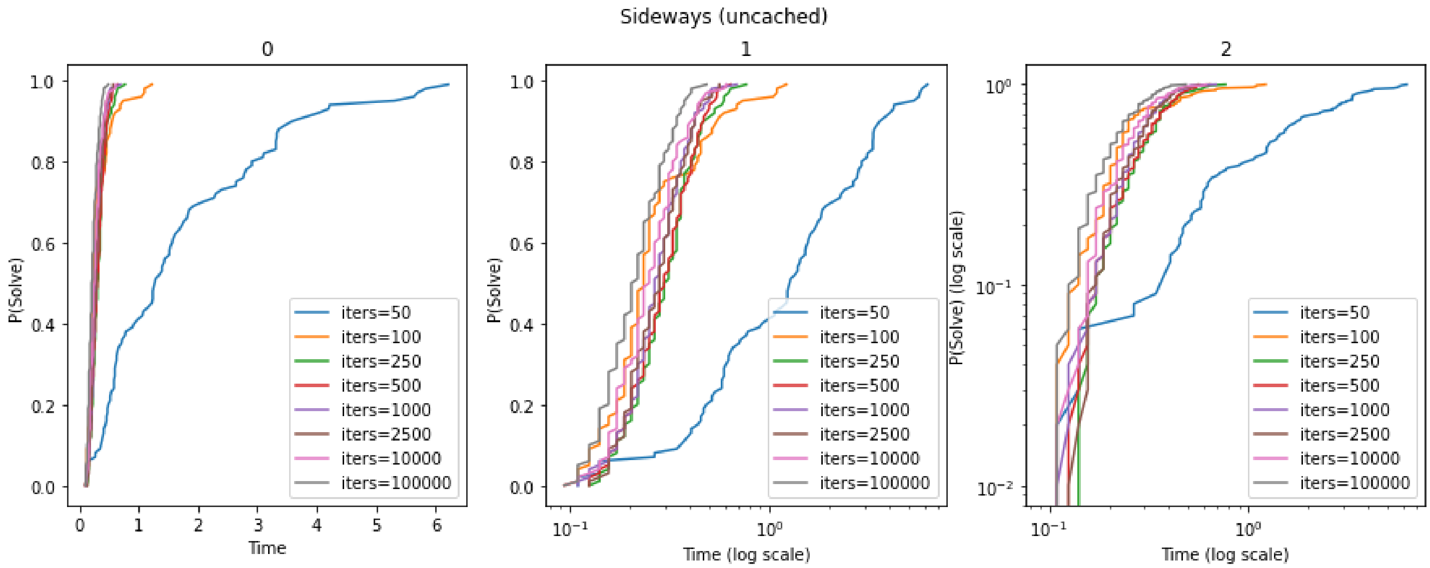
We observe that when iterations is set at 50, the performance is really bad and it takes a long time to reach a high probability of success. From earlier, we also saw that with 50, we never reach P(Solve)=100%. There is a marked increase when iterations is increased to 100, and then peaks out when iterations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

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

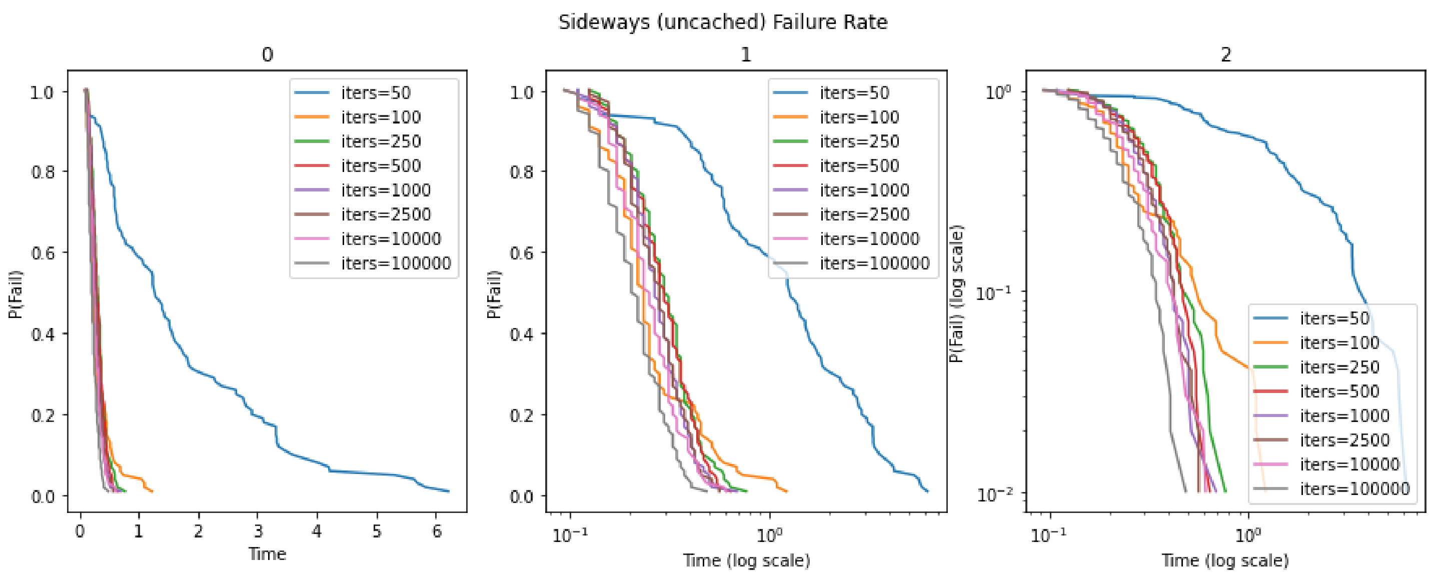
We see some variation between the different iteration limits beyond 250 in the run-time distribution, but when the run-length distribution is observed, we see that the number of operations beyond 250 is always the same. We can attribute the difference in run time to randomness and other factors.

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

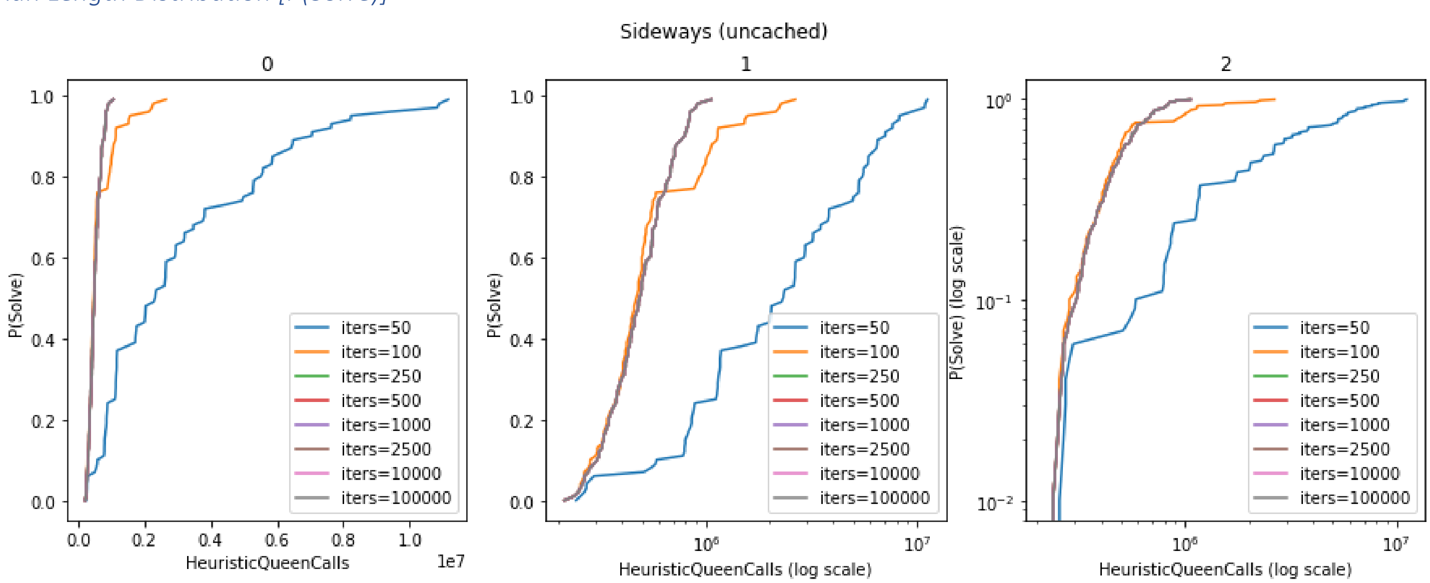
Run Time Distribution [P(Solve)]



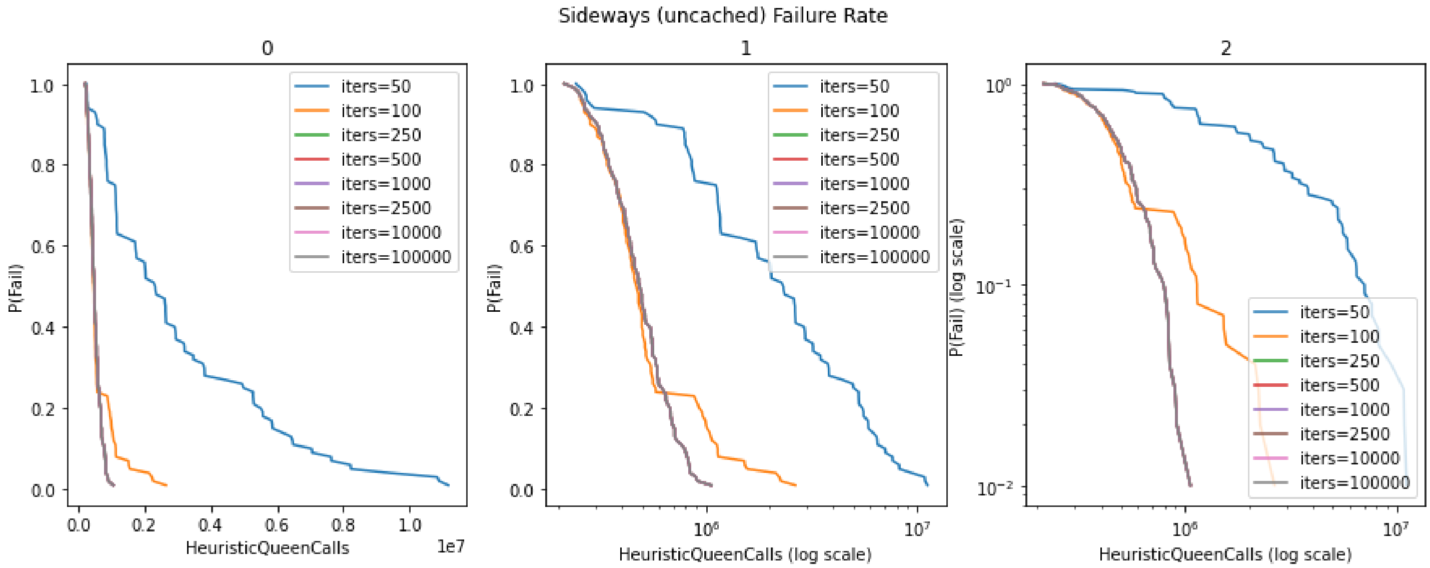
Run Time Distribution (Failure Rate)



Run Length Distribution [P(Solve)]



Run Length Distribution (Failure Rate)



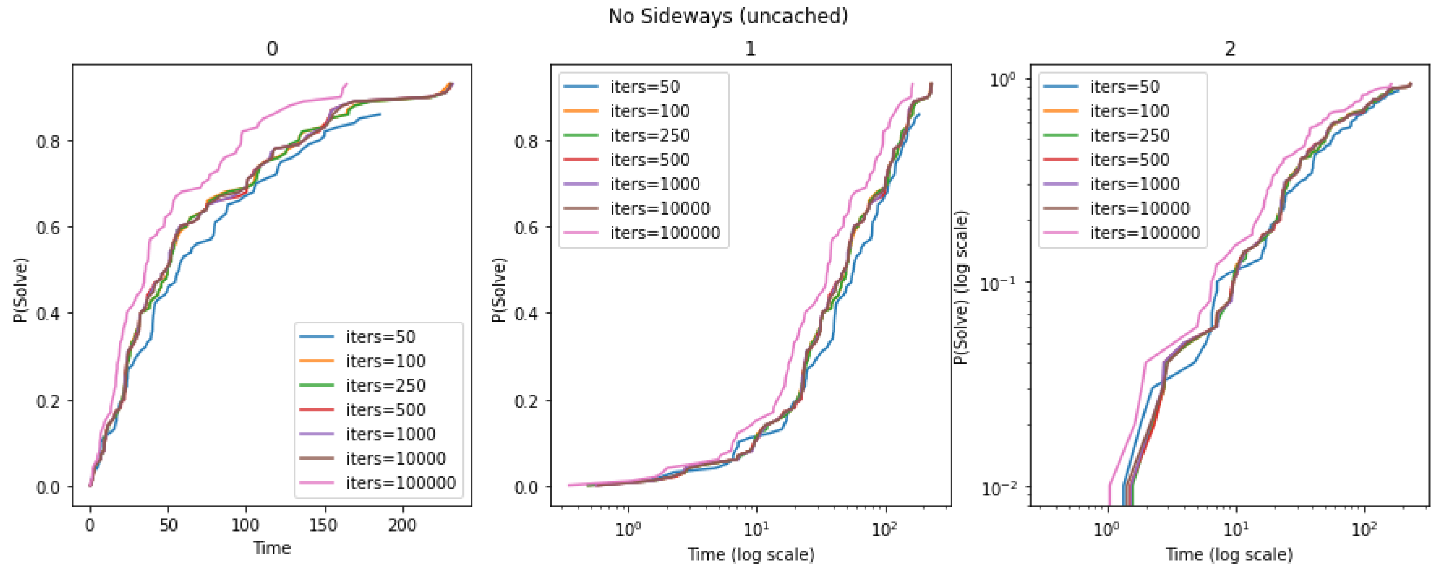
Sideways Not Allowed without Cache

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.

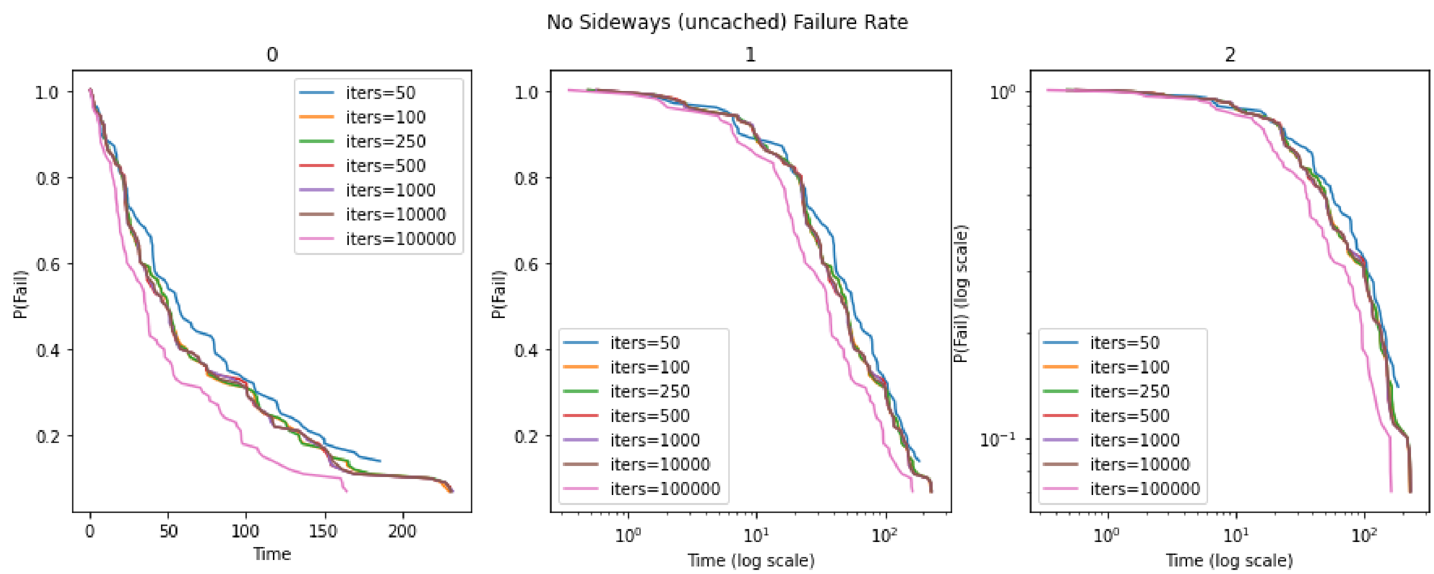
When we see the run-length diagram, we can see that the number of operations is almost exactly the same when the number of iterations is more than 100. The difference in time can boil down to random and other extraneous factors like machine load.

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.

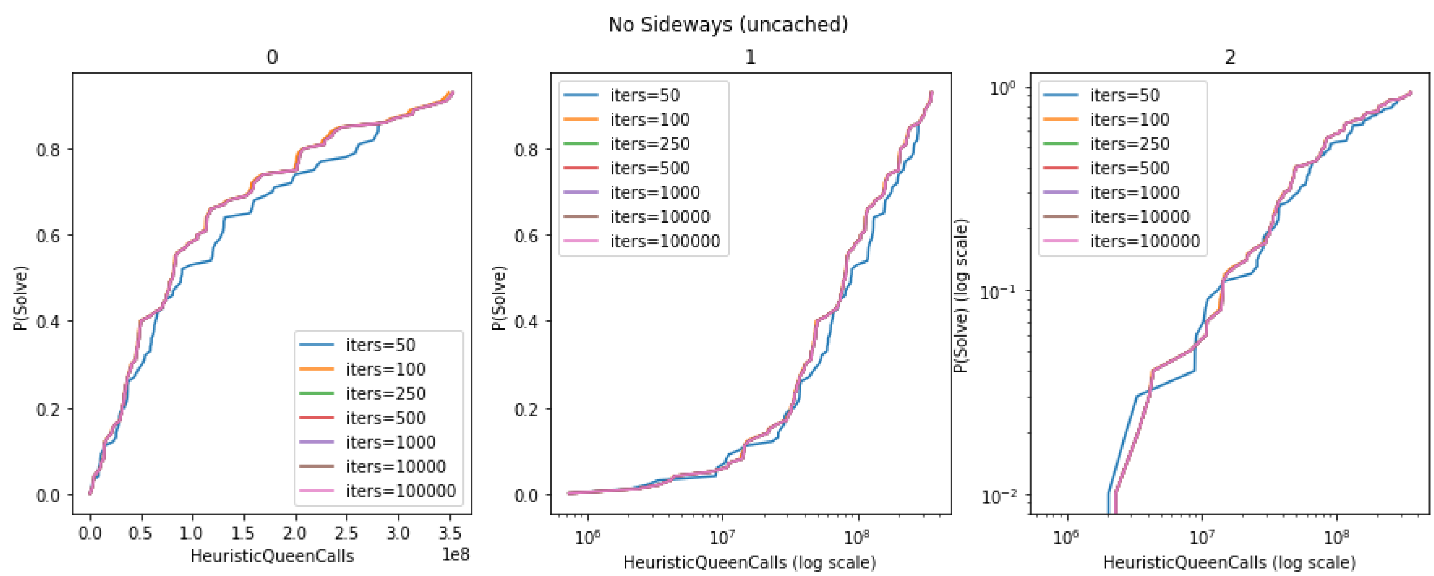
Run Time Diagram P(Solve)



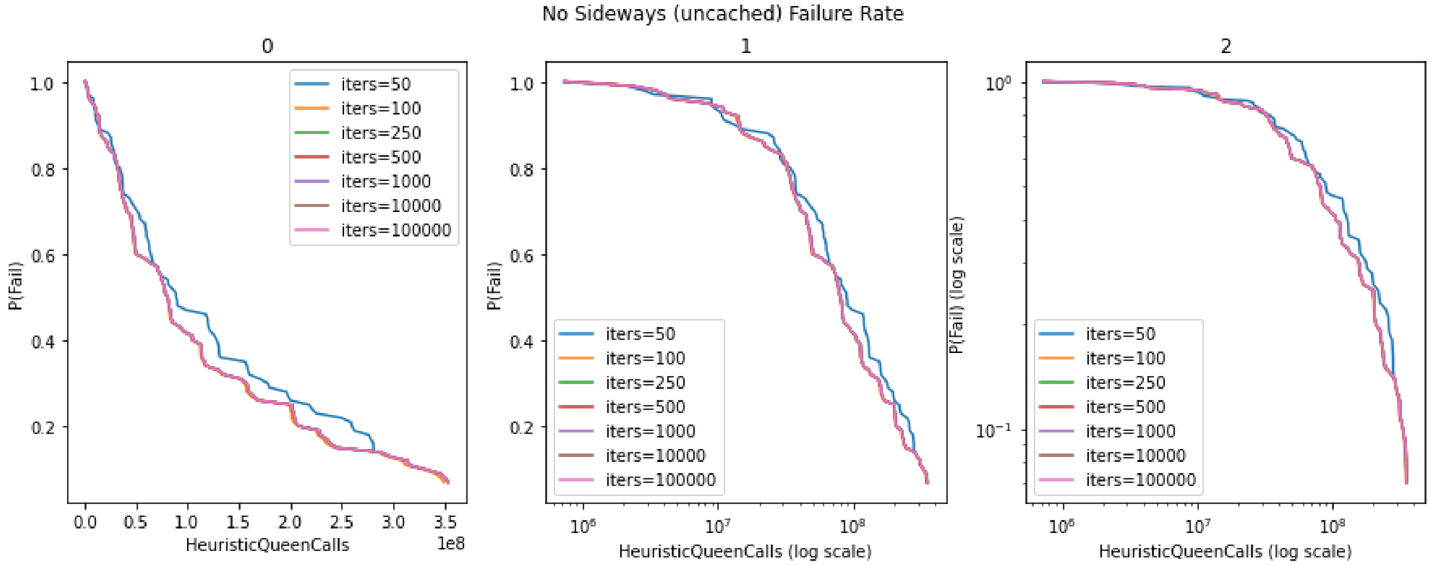
Run Time Diagram (Failure Rate)



Run Length Diagram P(Solve)



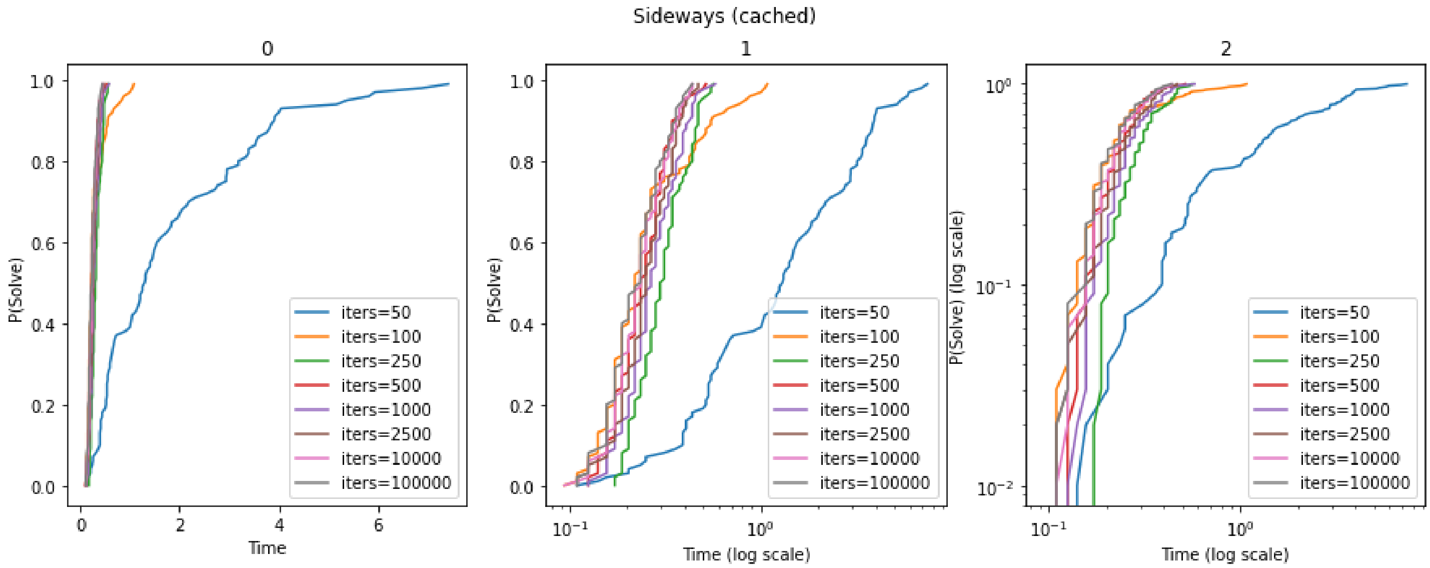
Run Length Diagram (Failure Rate)



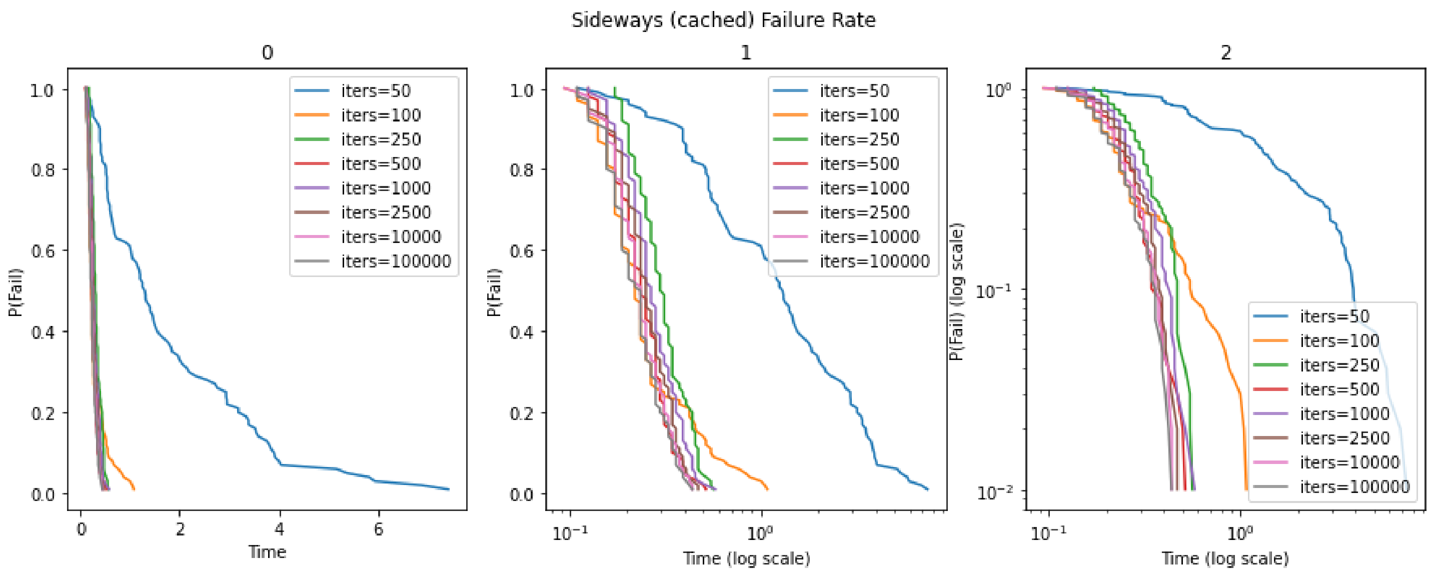
Sideways allowed with caching

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)



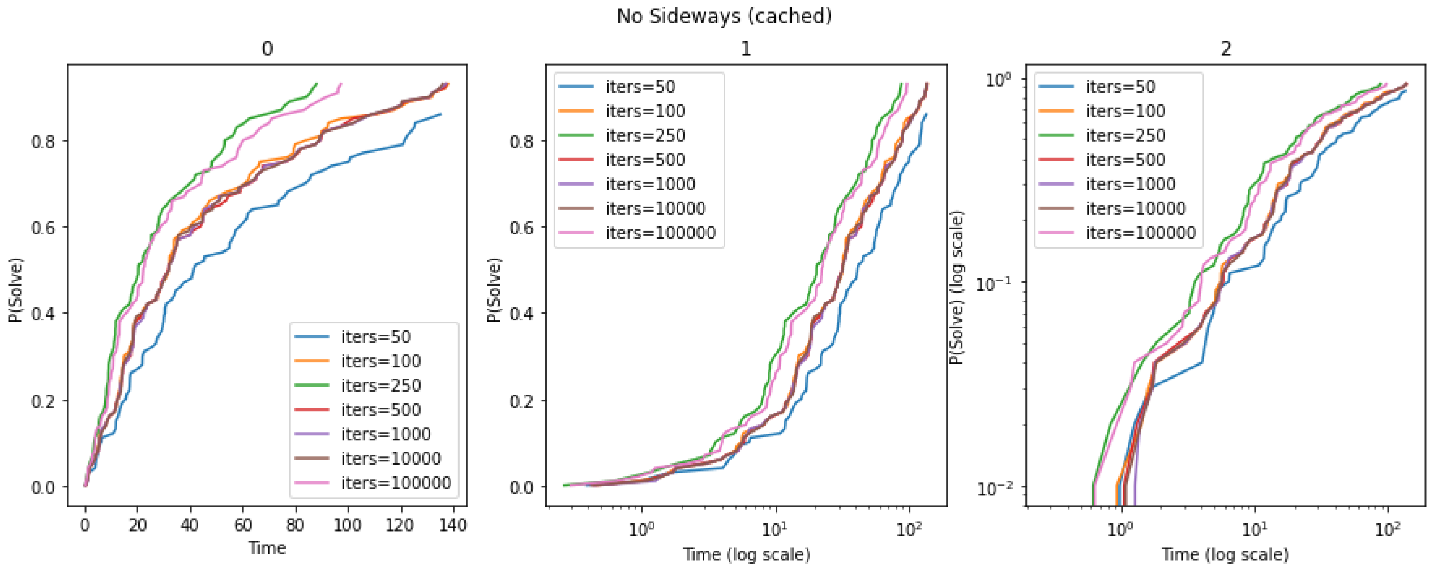
Run Time Distribution (Failure Rate)



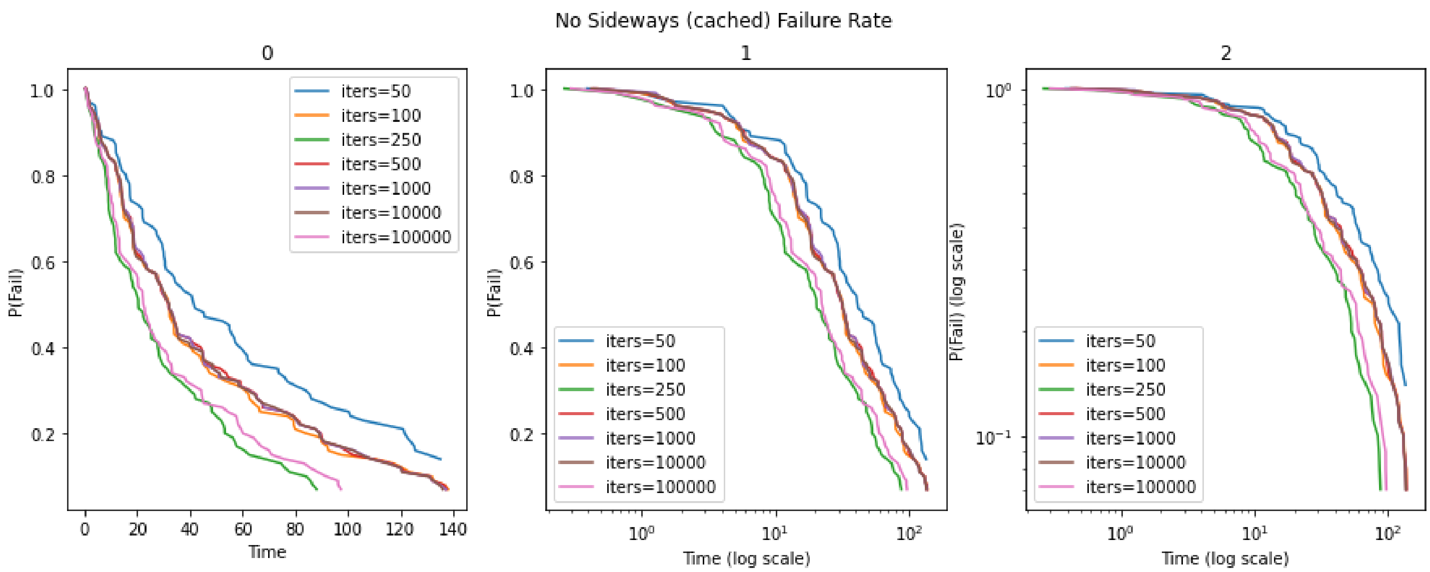
Sideways Not Allowed, cached

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)

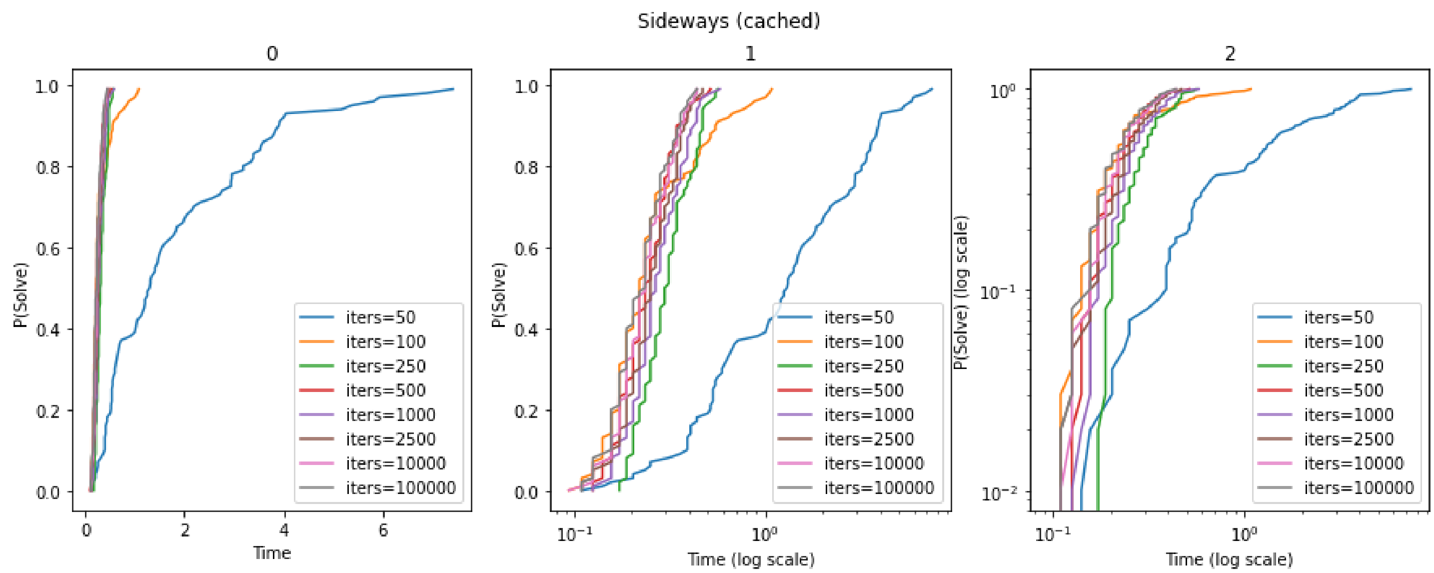


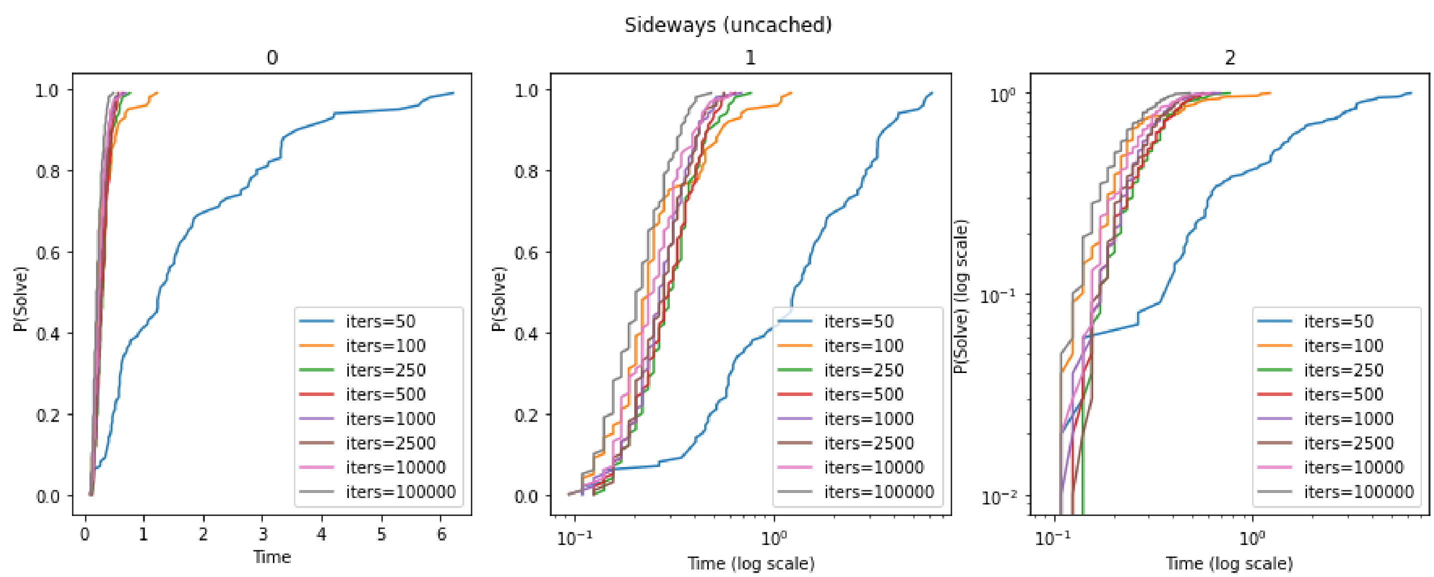
Run Time Distribution P(Fail)



Cached vs Non-Cached (sideways)

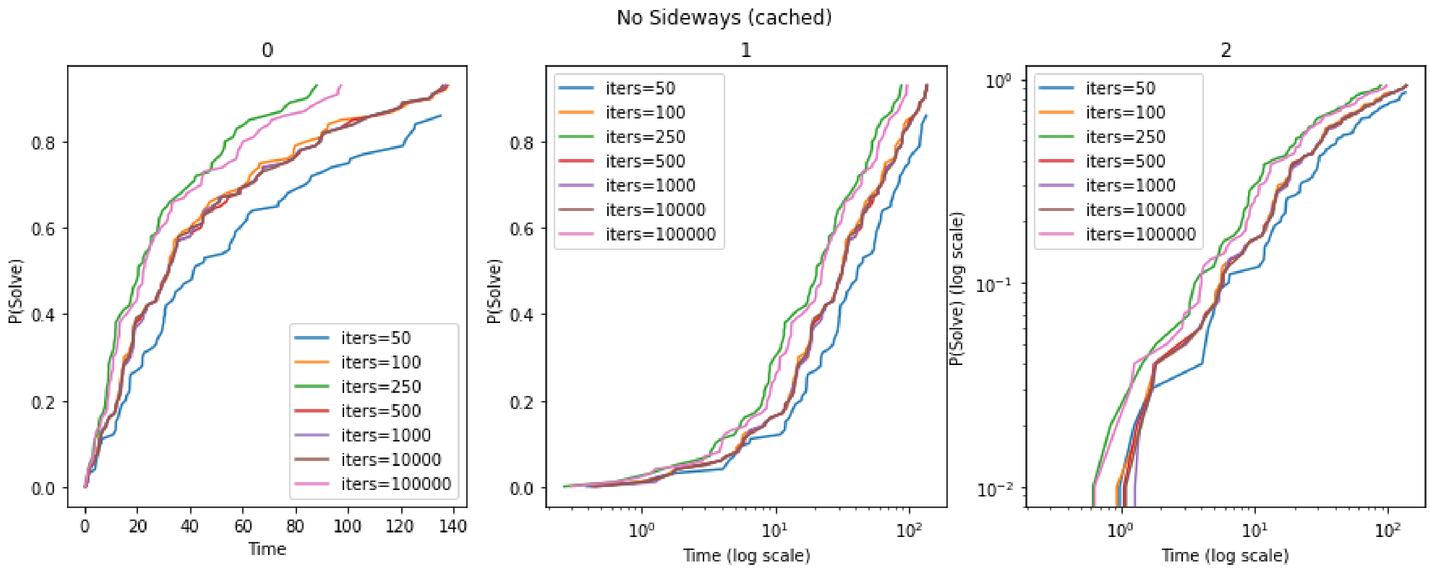
Here we observe that there is no meaningful change in the run time between cached and non-cached versions when sideways moves are allowed. This is because when sideways moves are allowed, things are quite fast by themselves, and the other overheads are more than the benefits from caching.

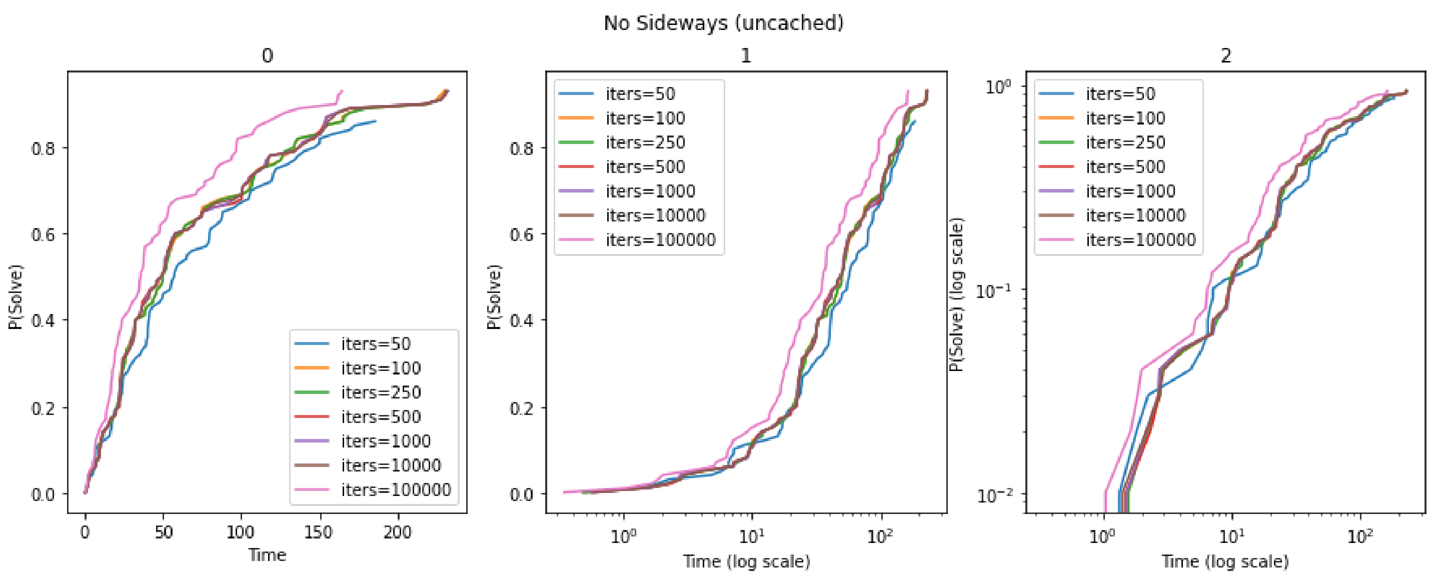




Cached vs Non Cached (Sideways Not Allowed)

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





Sideways vs Non-Sideways uncached

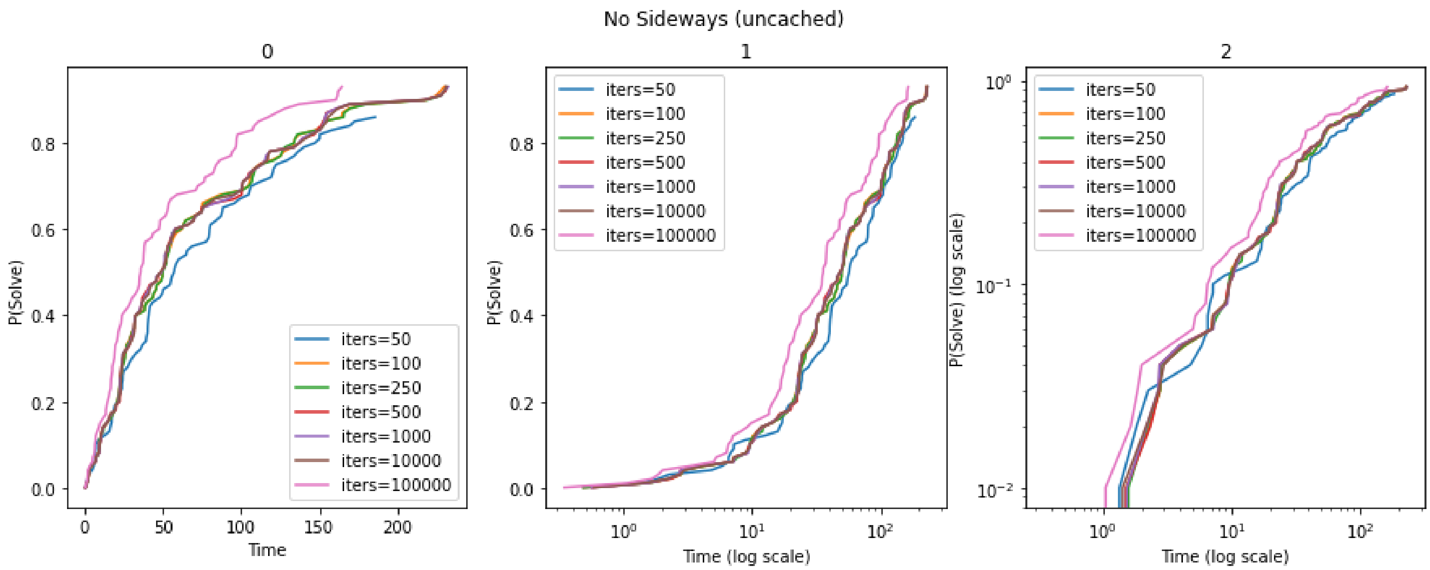
Here we see the following:

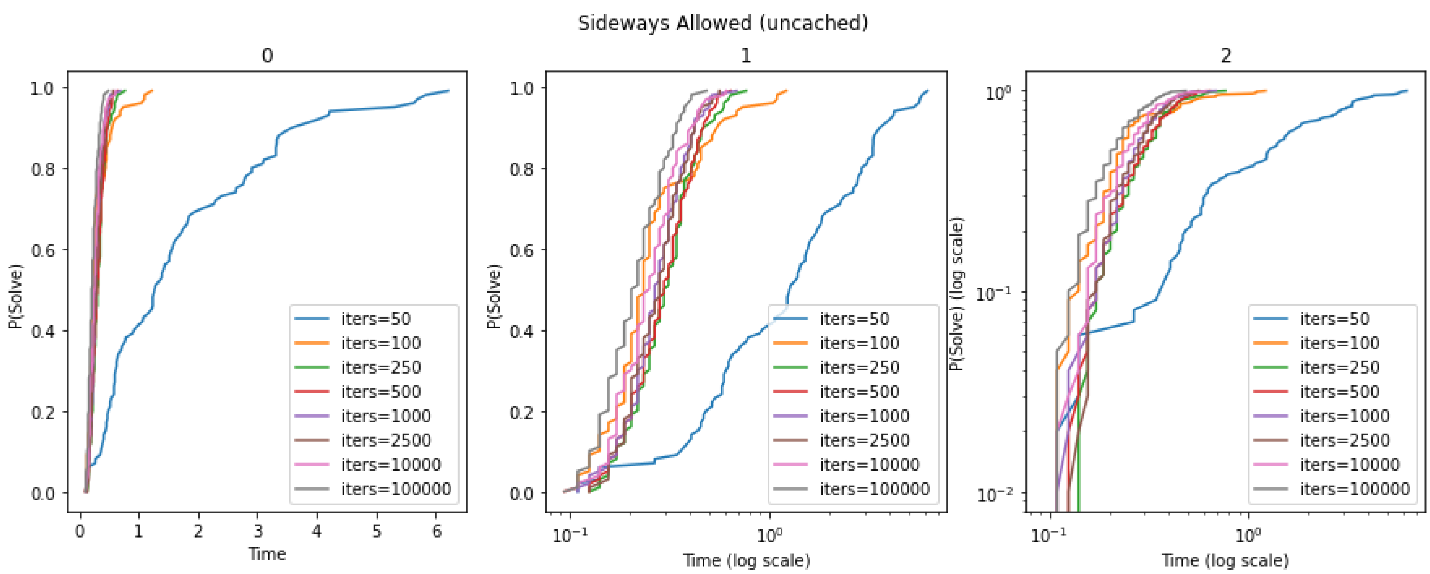
The version that allows sideways moves is significantly faster than the one that doesn't allow side moves

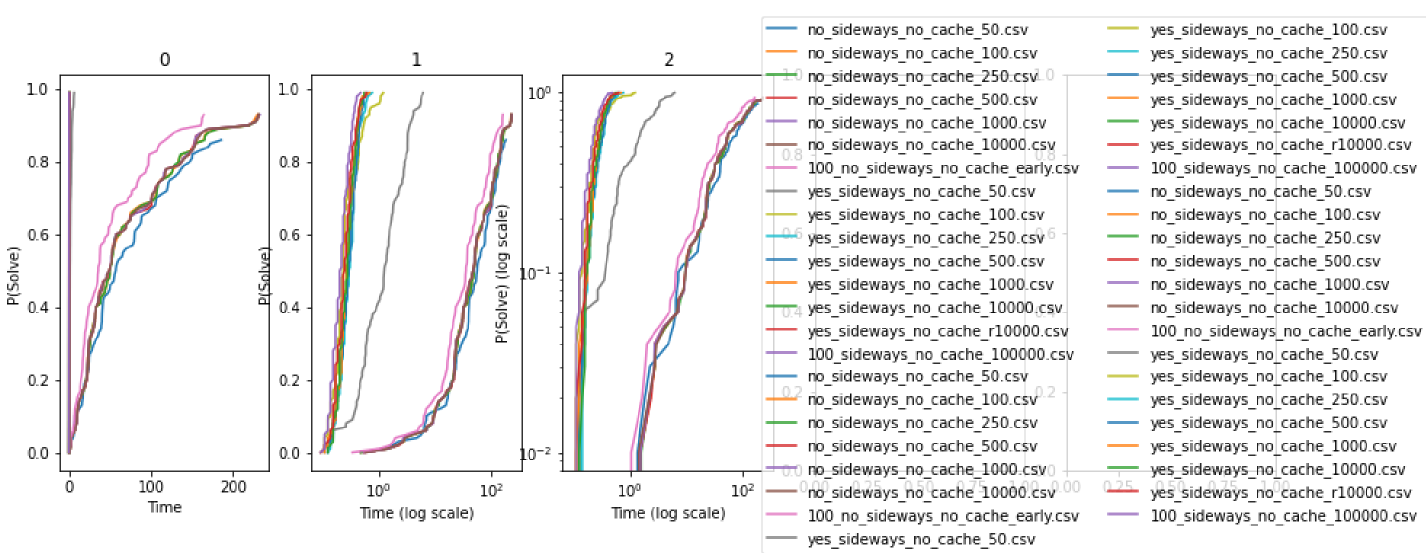
The version that allows sideways moves also has a bettes probability of success for both high and low number of iterations

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.







Conclusions and Key Takeaways

We have discussed the effects of varied approaches: introducing greater randomness, allowing non-improving moves, introducing caching, effect of early stopping criteria, and various methods for optimization. To summarize, these are the broad take-aways that we found were helpful in the implementation.

1. It is important to first profile the code, identify the functions that have the highest hit-count, and optimize those functions. Making each iteration faster allows us to have more iterations in a shorter amount of time, and thereby allowing us to get better solutions.
2. It is worthwhile to explore the possibility of using integer arithmetic wherever possible instead of floating point operations. In many algorithms, an approximation using integers can be good enough.
3. The use of simple arithmetic operations that map to primitive processor instructions (like add, multiply, etc.) perform much faster than some functions like pow() which use Taylor series expansion and are much slower. It helps if the latter are replaced with the former. In our case, we were able to replace the '**' operator which indirectly calls into pow() with the * operator which is a simple multiplication in distance calculation, and we got a significant boost in performance
4. Since each primitive can impact the total running time so extremely, it is worthwhile to consider whether implementation can be done in C instead of python
5. Evaluate if an early stopping criterion for each restart can be identified. This must only be done when it can be guaranteed that a local minima has been reached and no further iteration can (theoretically) improve the current restart. Stopping early can help save CPU cycles on non-improving iterations. This also allows us to set a very high limit on the iteration knowing well that if improvements are not possible, they will not be executed. The overall effect is that the run times goes down because non-improving iterations are avoided, and overall quality of results goes up because we can set a very high limit on the number of iterations without bothering about the consequences of setting them.
6. Caching can often yield a significant boost in performance. This is not always the case, but since implementation of caching in python is a single line of code using functools.lru_cache, it is worthwhile trying it out.
7. Introducing more randomness can yield faster results than a more exhaustive search. We see this in variant1 and variant2 of the TSP 2-opt local search algorithm.
8. Allowing non-improving moves can lead to faster results, and results of better quality. We see this in the N-Queens problem.
9. Doing n-runs in parallel, or n-restarts in parallel is also another way to make things faster.