Connor McManigal, mcmanigc (35523952)
Shuban Ranganath, shubanr (69643212)
Peyton Politewicz, ppolitew (21907867)

Team 4 CS271P Final Report

Branch and Bound Depth First Search Overview and Data Structures:

For our first approach to the Traveling Salesman Problem(TSP), we will be constructing a branch and bound algorithm using depth first search(BnB DFS). This algorithm is a generalization of depth first search that expands the deepest node and uses a lower bound paired with a heuristic function and an upper bound to prune the search space. With admissible heuristics, branch and bound typically has linear space complexity and exponential time complexity, which are nice properties to have in an algorithm. Some other key features are that it is an anytime algorithm, or it can be stopped and will still return some answer, and it is complete and optimal if run to completion with an admissible heuristic. Pruning of non-optimal branches in the search space is done by comparing an upper bound with a lower bound that uses a heuristic function. The upper bound imposes a depth bound and is used to indicate the current best known solution. Prior to running the algorithm, this bound is usually set to infinity. The lower bound is the minimum possible value that a solution to the problem can have within a specific subtree and this is calculated by the heuristic function. Pruning is performed when the lower bound is larger than the upper bound, and if this condition does not hold, then we keep the node and expand further. Lastly, since we are performing depth first search, we will be using a last-in-first-out queue or a stack for node storage and we will store nodes as objects.

Our problem instance is unique in that every node is connected to each other and for the TSP to be completed, we must finish the tour back home, or arrive back at the starting node. With that being said, the bidirectional exploration of the state space is a pivotal element in our algorithmic design. The state space provides a complete graph that lacks directionality, thus this implies that the path can be classified as bidirectional.

Our data structures for this implementation will include an adjacency matrix which serves as our search space, a stack or an array for storing node memory, heuristic values from our heuristic functions for each node, and each node will save the previous and successor node as a pointer. The problem generator that we have been provided with returns a nxn matrix where n is the number of nodes in the search space. This distance matrix represents a single instance of the TSP or a single search space to test the algorithm against. This generator allows us to create random instances of the TSP with varying number of nodes to visit, mean distances, and standard deviation of distances as controlled by n, mean, and sigma. The flexibility in creating instances allows us to generate a diverse set of TSP instances for testing and comparing algorithmic performances with different heuristic functions. We plan to instantiate our initial node, or starting point, by calculating the node with the lowest average distance to all other nodes. Our nodes will be stored as objects in an array, or a stack, and this will serve as a crucial element in managing the exploration of the search space. Each node will carry essential information about the permutation of nodes and the set of visited nodes. In order to ensure thorough exploration, as the algorithm continues to explore the state space, nodes are pushed onto the stack as new paths are discovered. All together, the array provides a structured means of retrieving and manipulating nodes in the search process, enforces modularity, and optimizes memory usage.

We will implement a simple zero heuristic to serve as our baseline, providing a reference for comparison. Our objective is to enhance the search process and enable more effective pruning, thereby increasing overall efficiency of our algorithm. These functions will be described in more detail in a later section.

BnB DFS Pseudocode:

```
Unset
#generate a randomized state space as an n x n matrix
problemMatrix <- createState(n, mean, std_dev)

#calculate the upper bound for the problem to help with pruning during BnB. We
use greedy search.

greedy_search(init_state, problemMatrix):
        cost <- NULL
        current_state <- init_state
        num_states <- problemMatrix.n #number of nodes in problem
        not_visited <- all unique nodes from problemMatrix
        not_visited.pop(init_state) #remove current state

        #iterate through unique nodes and find cheapest cost from start to finish
        while not_visited.not_empty():
                next_node <- sort(current_state.neighbours)
                if next_node in not_visited:
                        cost+= path_cost(current_state,next_node)
                        current_state <- next_node
                        not_visited.pop(init_state)
        cost+= path_cost(current_state,init_state)
        return cost

upperBound <- greedy_search()
p <- problemMatrix
```

```
BnB-DFS(p,f,upper_bound):
      var <- p.unassigned_variable()
      domain <- p.sort_child_nodes(var,f)
      stack.push(<var,domain>)
      best_assignment <- NULL
      U <- upper_bound
      while not stack.isEmpty()
            <var,domain> <- stack.top()
            if domain.isEmpty()
                  p.unassignvar(var)
                  stack.pop()
            else
                  value <- domain.pop()
                  p.assignvar(var,value)
                  if cost(p) >= U
                        continue
                  if p.hasFullAssignment()
                        U <- cost(p)
                        best_assignment <- p.currentAssignment()
                  else
                        var <- p.unassigned_variable()
                        domain <- p.sort_child_nodes(var,f)
                        stack.push(<var,values>)
      return best_assignment
```

BnB DFS Heuristic Function Pseudocode and Explanations:

We first begin by running a greedy search on the problem state space to find the upper bound by which our BnB is bound. This is a cheap heuristic that tells us what the optimal solution's highest cost should or would be. We run the greedy search by simply finding the lowest-cost unvisited neighbor and building a cyclical path that ends at the initial node. The resulting cost from the greedy_search() function is then used as an upper bound for the BnB-DFS function.

BnB DFS Pseudocode Explanation:

In the BnB-DFS function, we take the bounds with the problemMatrix provided to conduct a depth-first search into possible solution sets. We then create a stack to hold a history of nodes considered in a solution. As the DFS search progresses if any path's cost exceeds that of the upper bound, the branch is pruned, and the parent nodes' other children will be explored. Through each iteration where the goal state is reached, the Upper bound is updated with the lower path cost, and the search stack is reinitialized. This closely follows the logic of the BnB DFS shown in slides and adapts the use of greedy instead of function *f* for heuristics.

Evaluation of Time and Space Complexity:

The time and space complexity would be a sum of the complexities for greedy search and BnB DFS since they are not run in parallel.

The time complexity of Greedy is linear since it makes only one decision per node in the search space. However, the space complexity would be O(1) since the neighbor of each node and their associated cost are already given in the problem.

The time complexity of BnB DFS is exponential since we generate a graph of all possible paths in a fully connected graph where the number of edges increases quadratically, leading to an exponential increase in the number of possible paths in the worst case. However, the space complexity is linear since adding each node does not mean that the resulting new paths from the node are explored. We instead search with bounds that prune and ensure that the space occupied by the stack scales with the paths explored, which is linear.

Time complexity = O(b^d) where b is the branching factor and d is the depth

Space complexity= O(d) where d is the depth

Stochastic Local Search Overview and Data Structures:

Additionally, we will construct a stochastic local search(SLS) to address the traveling salesman problem. In local search optimization problems, the specific path taken to reach the goal is irrelevant and the goal itself is the solution. A state in a local search algorithm corresponds to a full path or solution. Typically, local search algorithms excel in handling large scale problems and yield satisfactory solutions but not always optimal ones. This algorithm aims to identify a complete configuration or path that satisfies certain predefined constraints while optimizing a cost or value. One key advantage of this algorithm is that it has the potential to be very memory efficient, as it maintains a small set of states. While traditional local search algorithms often encounter optimization challenges in regards to local minima and maxima, our approach of developing a SLS can strategically circumvent this issue. In the case of the TSP, we will use SLS to effectively minimize path cost and we will leverage randomness to avoid getting trapped in local minima. SLS algorithms determine the neighborhood of a solution by identifying states or solutions that are adjacent to the current known solution. The neighborhood directly influences the exploration of the state space and refers to the set of potential moves that can be applied to the current solution in the search process. Two common procedures of implementing SLS algorithms include greedy descent and random sampling. The greedy descent procedure is considered efficient but incomplete. It performs well when finding local minima, but performs poorly when exploring new paths of the search space. Greedy descent can be defined as the movement to a neighbor with the lowest cost. On the other hand, the random sampling method is complete but inefficient. It is well equipped for exploring new paths of the search space, but performs poorly when looking for local minima. A random restart wrapper may be used to improve the chances of locating an absolute minimum by reporting the best result found across many random restarts or trials. We propose that a combination of a greedy element and random restart will yield the most effective results. It should be noted that unlike branch and bound DFS, the starting node for an SLS algorithm is typically a random assignment.

Similar to BnB DFS, in the TSP, each node is connected to each other and for the problem to be solved, we must finish the tour back home. Again, the bidirectional exploration of the state space is a pivotal element in the design of our SLS algorithm.

Our data structures and functions for the SLS algorithm include the nxn tour weight matrix calculated by the problem generator, a list of standard deviations for each node calculated from the matrix, a list of size n to store the current path ordering, a tour generator function that randomly generates node orderings or paths, a tour calculator function that calculates the path cost from the current path and weight matrix, and a hill descent function. The tour weight matrix represents a single instance of the TSP or a single search space to test the algorithm against. The tour calculator will utilize the weight matrix and the respective arcs of the current path. Given a tour ordering and its associated arcs, the tour calculator will be instantiated as an nxn matrix that assigns 1's to the arcs corresponding to that specific path(i.e. arc 1→4 will assign 1 to [1,4]) and 0's to the arcs not involved in the current path. This matrix will be multiplied by the original tour weight matrix and will be summed to find the total path cost. The current path, as determined by the tour generator function, list of standard deviations, and nodes with the highest cost will be used by the hill descent function to swap the node with the highest cost with the node containing the lowest standard deviation. The list of the final path ordering will be determined by the hill descent function which swaps nodes based on the results of the path cost.

SLS Pseudocode:

```
Unset
#generate a randomized state space as an n x n matrix
problemMatrix <- createState(n, mean, std_dev)

#store all nodes in a list for manipulation
nodeList <- createList(loop n)

#calculate standard deviation of every node's set of path weights; store in
list where individual node n's standard deviation is in cell n-1.

list deviations <- calculateStdDev(nodeList)

#initialize to a random starting state
bestTour <- currentTour <- shuffle(nodeList)

#calculate path cost by generating nxn matrix of coordinates
```

```
pathMatrix <- coordMatrix(currentTour)
bestCost <- pathMatrix * problemMatrix


#local search
loop do
      if (iterations >= 4) then return bestFound
      else
            #swap order of lowest std. dev. node and node with highest
preceding value
            currentTour <- swap(currentTour)
            pathMatrix <- coordMatrix(currentTour)
            if pathMatrix * problemMatrix >= bestCost
                  restart
                  ++iterations
                  else
                  bestCost <- pathMatrix * problemMatrix
                  bestTour <- currentTour
```

<u>SLS Pseudocode Explanation:</u>

Our SLS approach is a modified version of the hill climbing algorithm. The complexity of calculating the neighborhood around a given tour solution seemed daunting given a scaling, fully connected node network. That is - ensuring we had identified and calculated every adjacent solution to a given tour seemed computationally intensive and difficult to ensure. As a result, we propose a potentially effective, lightweight solution that uses some preemptive analysis to identify nodes most- and least-likely to impact the path cost of a tour if their ordering is changed.

Our pseudocode walks through the process as follows:

- (A) We accept the *n* x *n* problem matrix from the generator.
- (B) We create a list of *n* nodes to manipulate for path creation.
- (C) We calculate the standard deviation of every node's path and store it in a list that mirrors B.
- (D) The list from B is shuffled to create a starting point and this is set as the best known tour.
- (E) The tour path from D is utilized to create a second matrix that is then for multiplication and calculation of path cost.
- (F) The node with the minimum value from C and the node with the longest preceding path weight from E are swapped in the tour ordering. **If** there is an improvement, repeat. If not, add the path to a record, then restart with a new randomly generated tour. Repeat until four iterations have completed.

SLS Objective Function Details:

The objective of the SLS function is hill descent. The peculiarity we're aiming to exploit can be exemplified by the following example:

Imagine some hypothetical tour with five nodes, completely connected. One node, Q, has path weights of [11, 13, 14, 9] to the other nodes. Allow another node, R, to have path weights of [8, 11, 19, 22]. Even in its worst case, Q can only increase the cost of a tour by 5. On the other hand, R can potentially switch from 22 to 11; an 11-point cost savings.

Our thinking is that in many tours, there will be at least one node with relatively low standard deviation across its path weights such as Q above. We propose using this node to aggressively cycle through its partner spaces, treating it as a 'free space' to see if we discover substantial reductions in total path cost by exchanging this relatively stable node with the highest cost node of a tour.

After four iterative resets and attempts to find local minima, we disengage the algorithm. This is a greedy approach - it doesn't try to do anything beyond making an improvement on its immediate next move.

Evaluation of Time and Space Complexity:

Relatively lightweight for required space, seemingly - two matrices that scale with n, two lists that scale with n. Both linear. Only a few previous states are saved in memory.
Time is also gated solely by n.

Results and Conclusions:

To assess the performance of our Branch and Bound Depth-First Search and Stochastic Local Search algorithms, we used the following framework. For each algorithm, we tested four different cases: small search space with low standard deviation, small search space with high standard deviation, large search space with low standard deviation, and large search space with high standard deviation. Testing our algorithms on our laptops proved to be difficult, perhaps due to limited processing power, thus we experienced issues in regard to running TSP instances with very large search spaces. Therefore, we defined the small search space as 10 nodes and large search space as 50 nodes. We held the mean distance of nodes as a constant of 20 across all tests. Next, we defined low standard deviation as being approximately 25%(i.e. 5) of the mean and high standard deviation being 100%(i.e. 20) of the mean. We thought that this would serve as an adequate and thorough approach to testing the strengths and weaknesses of each algorithm.

Branch and Bound DFS:

Through our experimentation we see that branch and bound uses almost half the memory of SLS to operate on smaller TSP instances given its relatively lightweight heuristic and storage usage. As we mentioned earlier, space complexity scales linearly with the depth of the recursion stack or O(d), which aligns with our results when comparing BnB and SLS. We also notice that

BnB does return the most optimal path, making it the more efficient algorithm when paired with the greedy heuristic. The tour cost is approximately 3 units smaller than the SLS algorithm in small search space with low standard deviation and approximately 8 units shorter than SLS in the small search space with high standard deviation. Also, we should note that the BnB algorithm performs worse with a path weight of 158 compared to SLS's path weight of 150 on the same graph with a small search space and low variance, but provides a smaller path weight of 91 compared to SLS's path weight of 96 when the search space is small and standard deviation is high. This could possibly suggest that branch and bound is better in regards to path weights for high standard deviation instances compared to low standard deviation search spaces.

We can also see that the BnB struggles in terms of runtime when run on low deviation graphs where it takes 212x more times to return a path. However, in the small search space with high standard deviation, the runtime is about half a second longer than its counterpart SLS algorithm. At its worst, time complexity for a branch and bound algorithm is exponential in terms of the depth measured as O(b^d). This may suggest that runtime for branch and bound is sensitive to problem characteristics such as standard deviation. In other words, BnB may struggle to efficiently prune unpromising branches in a search space where standard deviation is low. Additionally, our branch and bound algorithm outputs surprisingly different results in regard to the number of branches pruned. In the low deviation search space, a total of 1.7 million branches were pruned while in the high deviation space only 63,000 branches were pruned. Perhaps the BnB algorithm had a more difficult time pruning in the low standard deviation than the high standard deviation search space. This result further supports our claim that standard deviation is a significant factor in determining the algorithms runtime and pruning abilities. Despite the fact that our BnB algorithm has a longer runtime, in small search spaces, it consistently provided a more optimal path with a lower tour cost.

In terms of larger graphs, with what we have defined as a large search space(i.e. 50+ nodes), we see the BnB algorithm times out while keeping low memory usage. Due to the timeouts, however, we cannot make any solid conclusions about BnBs performance on large search spaces. We believe that if we had stronger computing abilities(more than the power of a standard laptop), that the branch and bound algorithm would be able to run thoroughly. This caused us to raise questions about branch and bounds possibilities for exponential time complexity. We believe that these timeouts could have been caused by the depth of the search space and the resulting large branching factor. If this was the case, the algorithm would explore an exponentially increasing number of nodes as the search space expanded.

All together, we witnessed that our branch and bound algorithm demonstrated its capability to deliver optimal solutions in scenarios characterized by a small search space. However, this precision in solution quality came at the expense of runtime, with BnB exhibiting longer execution times in comparison to SLS. Notably, the peak memory usage of BnB was about half of SLS and this observation underscores the inherent tradeoff between time and space efficiency in algorithmic performance.

Stochastic Local Search:

SLS has some of the lowest runtime when running in small search spaces regardless of deviation, showing that deviation of a set has relatively little effect on its ability to find close to optimal solutions. But, some of the downsides of this algorithm include not being able to detect when no solution exists. While its memory usage compared to BnBs ~8500B at its peak is relatively high, it remains consistent. Unlike BnB, we see SLS algorithms perform slightly better with low standard deviation sets. SLS's algorithm makes it a memory burden on smaller datasets but its efficiency of memory usage is shown on much larger datasets where it also provides a significant speedup

While using SLS on larger search spaces, we notice a linear scaling of time and memory. With a 5 fold increase in dataset size we notice a ~30% increase in peak memory usage and a 10x increase in time in both the low and high standard deviation datasets. This also shows that variance and standard deviation have a slight effect on SLS regardless of search space size. The nature of SLS also means that it can return an optimal path whenever one is available at a much quicker rate than BnB on similar search spaces. Our heuristic for BnB uses a similar greedy strategy to help prune branches but SLS's ability to optimize greedy to return the optimal path makes it the algorithm of choice for large datasets with indeterminate variances. While we do see a lot more iteration on SLS based on the variance of the path weights such an increase does not reflect on the memory usage and instead increases the runtime by a large amount as noticed by the 0.05 to 0.9 on smaller search spaces and ~0.4 seconds for both large search spaces.

We also observed that high standard deviations lead to more iterations on smaller search spaces, but inversely larger search spaces have a decrease in iterations with a rise in deviation. We believe that higher deviations provide a greater ability to explore better paths at the start of the process which is very helpful for finding better paths earlier whereas smaller search spaces pay the price for this early optimization but do not run long enough to reap the benefits.

Conclusion and Proposed Improvements:

Through this project, we observed a variety of differences between the performances of these two algorithms and witnessed the time and space complexity tradeoff. In small search spaces, BnB provided a more efficient and optimal solution with less memory usage, but at the cost of longer run times. On the other hand, SLS provided a near optimal solution with faster runtime, but with almost double the memory consumption. In larger search spaces, BnB failed to run to completion and timed out, presumably due to a large branching factor and depth of the TSP instance. But, in the larger search space, SLS succeeded in providing a fast solution at the cost of large memory consumption.

In regard to improving the branch and bound algorithm, a variety of strategies could be explored and implemented. Some possible improvements could include attempting new heuristic functions or bounding strategies to enhance the effectiveness of pruning, or implementing an early stopping criteria to halt the algorithm when a satisfactory solution is found.

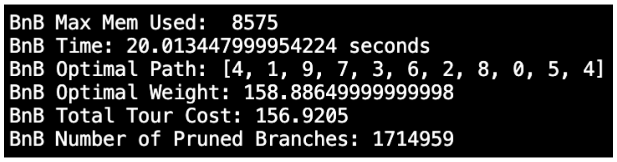
To improve our SLS algorithm, we could increase the number of allowed iterations, thus permitting additional randomness which could lead toward a more optimal solution.

Overall, this project could further be enhanced by a few considerations including, using a more efficient programming language that can release unnecessary memory and utilizing more powerful hardware or cloud computing resources that would provide more computational power.

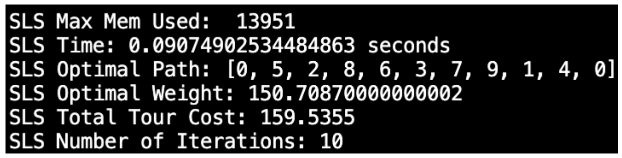
We are also limited by the tools available when measuring memory usage and runtime where each run is highly variable and dependent on the OS and runtime environment of the PC. With a more accurate way to measure memory usage and the ability to ensure no other processes infringe on the runtime, we can draw more solid conclusions about the actual resource utilization of each algorithm.

Appendix:

Small Search Space and Low Standard Deviation(10 nodes, 20 mean, 5 standard deviation):

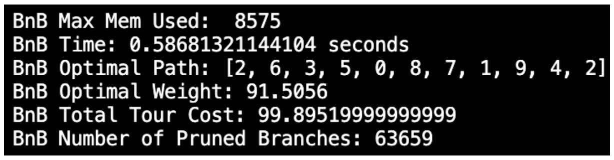


```
BnB Max Mem Used:  8575
BnB Time: 20.013447999954224 seconds
BnB Optimal Path: [4, 1, 9, 7, 3, 6, 2, 8, 0, 5, 4]
BnB Optimal Weight: 158.88649999999998
BnB Total Tour Cost: 156.9205
BnB Number of Pruned Branches: 1714959
```

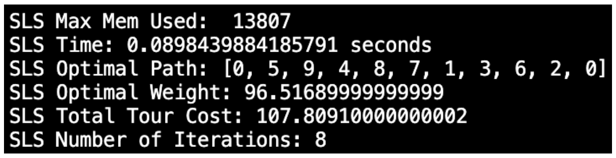


```
SLS Max Mem Used:  13951
SLS Time: 0.09074902534484863 seconds
SLS Optimal Path: [0, 5, 2, 8, 6, 3, 7, 9, 1, 4, 0]
SLS Optimal Weight: 150.70870000000002
SLS Total Tour Cost: 159.5355
SLS Number of Iterations: 10
```

Small Search Space and High Standard Deviation(10 nodes, 20 mean, 20 standard deviation):

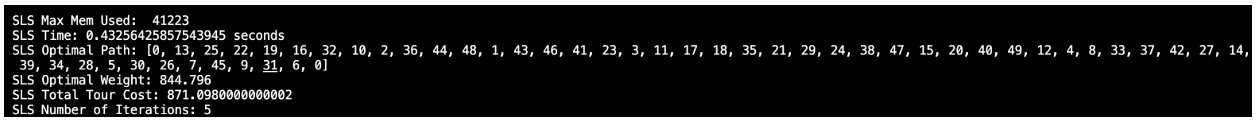


```
BnB Max Mem Used:  8575
BnB Time: 0.58681321144104 seconds
BnB Optimal Path: [2, 6, 3, 5, 0, 8, 7, 1, 9, 4, 2]
BnB Optimal Weight: 91.5056
BnB Total Tour Cost: 99.89519999999999
BnB Number of Pruned Branches: 63659
```



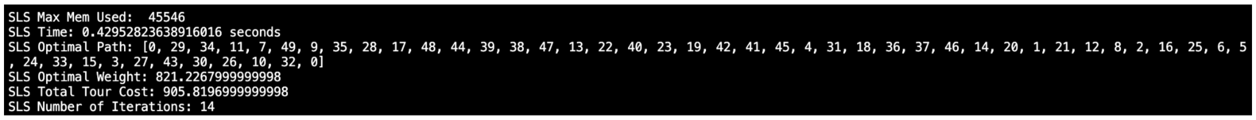
```
SLS Max Mem Used:  13807
SLS Time: 0.0898439884185791 seconds
SLS Optimal Path: [0, 5, 9, 4, 8, 7, 1, 3, 6, 2, 0]
SLS Optimal Weight: 96.51689999999999
SLS Total Tour Cost: 107.80910000000002
SLS Number of Iterations: 8
```

Large Search Space and Low Standard Deviation(50 nodes, 20 mean, 5 standard deviation):



```
SLS Max Mem Used:  41223
SLS Time: 0.43256425857543945 seconds
SLS Optimal Path: [0, 13, 25, 22, 19, 16, 32, 10, 2, 36, 44, 48, 1, 43, 46, 41, 23, 3, 11, 17, 18, 35, 21, 29, 24, 38, 47, 15, 20, 40, 49, 12, 4, 8, 33, 37, 42, 27, 14,
 39, 34, 28, 5, 30, 26, 7, 45, 9, 31, 6, 0]
SLS Optimal Weight: 844.796
SLS Total Tour Cost: 871.0980000000002
SLS Number of Iterations: 5
```

Large Search Space and High Standard Deviation(50 nodes, 20 mean, 20 standard deviation):



```
SLS Max Mem Used:  45546
SLS Time: 0.42952823638916016 seconds
SLS Optimal Path: [0, 29, 34, 11, 7, 49, 9, 35, 28, 17, 48, 44, 39, 38, 47, 13, 22, 40, 23, 19, 42, 41, 45, 4, 31, 18, 36, 37, 46, 14, 20, 1, 21, 12, 8, 2, 16, 25, 6, 5
, 24, 33, 15, 3, 27, 43, 30, 26, 10, 32, 0]
SLS Optimal Weight: 821.2267999999998
SLS Total Tour Cost: 905.8196999999998
SLS Number of Iterations: 14
```

SLS Results, sorted by alphanumeric descending source file name:

4		
21907867,69643212,35523952		
SLS		
0.08414196968078610	,	8563.512135964320
0.08014583587646480	,	9710.336621954220

0.08751797676086430	,	9200.633752584830
0.08542227745056150	,	9726.416466026460
0.08783912658691410	,	9085.018153691450
0.08160519599914550	,	9729.634825156100
0.08444333076477050	,	9034.463572204750
0.08210611343383790	,	9773.828023719670
0.08383607864379880	,	9089.133158716220
0.08564519882202150	,	9742.272287514330
3.5617241859436000	,	97352.46545254830
3.629826068878170	,	99170.54523689780
3.7466979026794400	,	97907.73296058760
3.5453109741210900	,	99285.11356032990
3.635708808898930	,	97223.74764672770
3.560626983642580	,	99429.37841522060
3.597975730896000	,	97424.86845237990
3.6240270137786900	,	99338.72983750120
3.6429638862609900	,	97273.34809321640
3.5258262157440200	,	99372.67630961470
0.22033381462097200	,	18728.2643937522
0.21325063705444300	,	19666.777426170000
0.23176193237304700	,	18690.99618645620
0.21109986305236800	,	19637.451455396500
0.22684216499328600	,	18902.135430571200
0.21100401878356900	,	19750.452992952200
0.21971702575683600	,	18618.11439842310
0.21021795272827100	,	19644.109927463600
0.2238929271698000	,	18917.818186703900
0.20788979530334500	,	19698.018777575100
0.015893936157226600	,	2049.1616014632600
0.015804290771484400	,	2327.437510979620
0.015970945358276400	,	2040.6292636335900
0.016060829162597700	,	2330.644650790500
0.01624917984008790	,	1843.2523499322600
0.015357017517089800	,	2316.7256619833300
0.015662193298339800	,	1840.0825654887000
0.01429605484008790	,	2424.8458511690000
0.015949010848999000	,	2027.5259292449000

0.015969038009643600	,	2334.762450419390
0.4375269412994390	,	28331.040298032100
0.4155881404876710	,	29684.19491463280
0.4288458824157720	,	28219.657380546300
0.418651819229126	,	29540.32008341940
0.4295330047607420	,	27722.147535673100
0.4201829433441160	,	29505.213441217600
0.4327239990234380	,	28410.131927475400
0.4193267822265630	,	29552.465174220700
0.4307117462158200	,	28322.909796437700
0.42989015579223600	,	29585.842373484900
0.7125427722930910	,	38386.57881267190
0.7198500633239750	,	39644.02318104120
0.7216567993164060	,	38507.253590806700
0.7131509780883790	,	39373.35219172100
0.7238621711730960	,	38248.85692522400
0.721153974533081	,	39562.38272590830
0.726423978805542	,	37961.67811179310
0.7154660224914550	,	39603.15766360600
0.7396080493927000	,	37703.63534952840
0.6983461380004880	,	39599.99964566930
0.034842729568481400	,	4382.554900323600
0.03582191467285160	,	4820.799149441480
0.03620409965515140	,	4703.885559102450
0.03583979606628420	,	4701.175444990660
0.03512907028198240	,	4164.802531773960
0.03174710273742680	,	4828.7229684416900
0.037011146545410200	,	4167.150262977490
0.0342707633972168	,	4735.855145132810
0.03578996658325200	,	4273.836716989710
0.03507494926452640	,	4788.2583983448900
1.4724440574646000	,	58096.564895535200
1.4480857849121100	,	59561.016319949600
1.4837219715118400	,	57582.80095699660
1.4395570755004900	,	59500.748454463400
1.469905138015750	,	57976.920139436200
1.4491171836853000	,	59415.2425226822

1.4592907428741500	,	57922.954799206100
1.445741891860960	,	59410.06831424010
1.5125329494476300	,	57388.723296690200
1.4533400535583500	,	59517.78952304760
0.058302879333496100	,	6938.7366733731500
0.05797410011291500	,	7261.040656115950
0.0587921142578125	,	6792.611370369090
0.05733299255371090	,	7243.928218128630
0.05996417999267580	,	6714.960450436990
0.05577802658081060	,	7232.41220106855
0.05638599395751950	,	6865.144008231730
0.05791902542114260	,	7312.507295444600
0.058811187744140600	,	6787.742178924770
0.056034088134765600	,	7303.458431834000
2.467423915863040	,	77817.48926709360
2.4159719944000200	,	79469.37807781690
2.3981189727783200	,	77423.70414635690
2.431649923324590	,	79388.94835423800
2.449090003967290	,	77136.48764776190
2.3998682498931900	,	79445.08445733700
2.456801176071170	,	76426.0568298334
2.4026589393615700	,	79395.3386337784
2.4174787998199500	,	77719.36351135130
2.419355869293210	,	79376.43929931800