Report - Giorgia Modi

Laboratory 1

Repository: https://github.com/GiorgiaModi/CI2024_lab1

Set cover problem: Given a family of sets, find the subfamily of minimum cost that covers all elements in the universe. Several possible greedy algorithms.

Lab1 request: Solve efficiently these instances customizing a technique discussed in class

Instance	Universe size	Num sets	Density
1	100	10	.2
2	1,000	100	.2
3	10,000	1,000	.2
4	100,000	10,000	.1
5	100,000	10,000	.2
6	100,000	10,000	.3

I have implemented two optimization algorithms to solve the given problem:

- **Solution 1** is based on **Simulated Annealing**
- **Solution 2** is based on **Tabu Search**

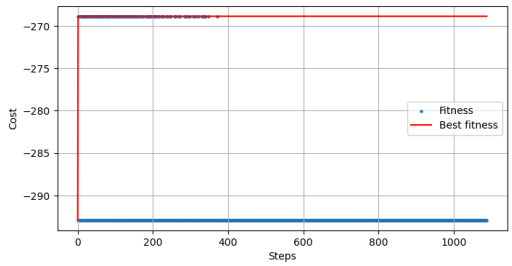
Results:

Instance	Universe Size	Num Sets	Density	Execution Time Sol 1	Execution Time Sol 2	Fitness Solution 1	Fitness Solutio
1	100	10	0.2	0.1s	0.6s	-301.7654322960134	-283.53
2	1,000	100	0.2	0.1s	1.6s	-7105.292537437867	-6045.4
3	10,000	1,000	0.2	1.1s	16.2s	-359025.4417471046	-19445
4	100,000	10,000	0.1	3m 45.1s	81m 32.1s2	-105543124.2185235	-15939
5	100,000	10,000	0.2	3m 56.6s	82m 6.8s	-231371537.8134606	-34334
6	100,000	10,000	0.3	3m 49.9s	87m 5.3s	-347324626.9237651	-56080

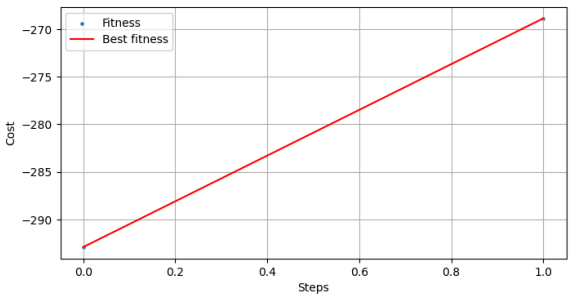
Based on the performance data in the table:

1. **Execution Time: Simulated Annealing (Solution 1)** is consistently faster than **Tabu Search (Solution 2)**across all instances. For smaller instances (Instance 1 and 2), the difference in execution time is relatively small, but as the problem size increases (e.g., Instance 4 and above), Tabu Search becomes significantly slower, with a runtime exceeding one hour on the largest instances.
2. **Fitness: Tabu Search (Solution 2)** produces better fitness values across all instances, indicating that while it is slower, it tends to find better solutions.

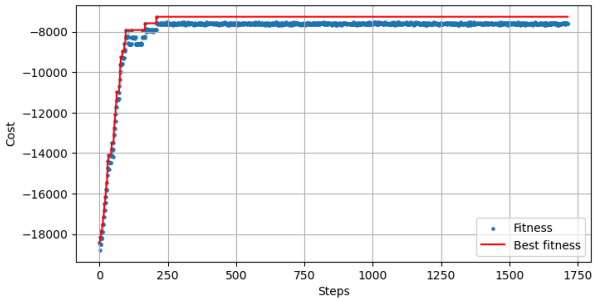
While Simulated Annealing handles larger instances more efficiently in terms of execution time, the trade-off is a lower fitness quality. Tabu Search, though computationally expensive, proves to be more robust in terms of solution quality, especially for larger and more complex instances.



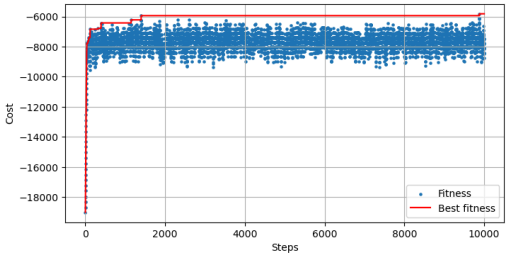
Instance 1 - Simulated Annealing



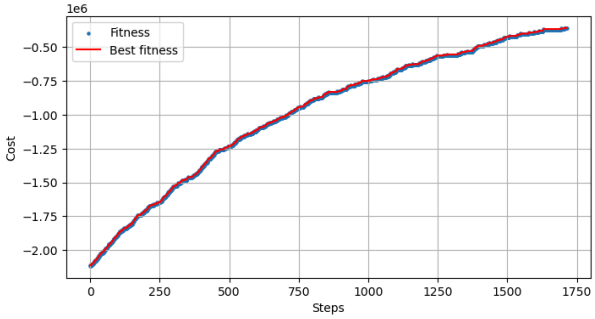
Instance 1 - Tabu Search



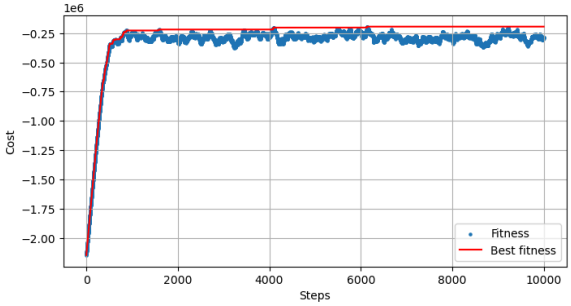
Instance 2 - Simulated Annealing



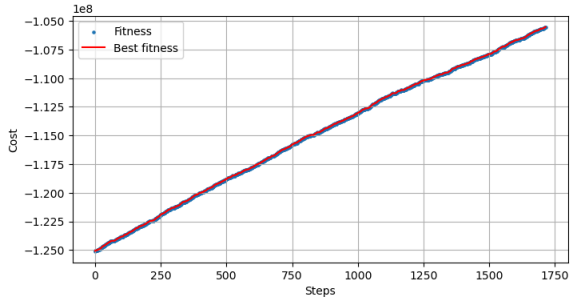
Instance 2 - Tabu Search



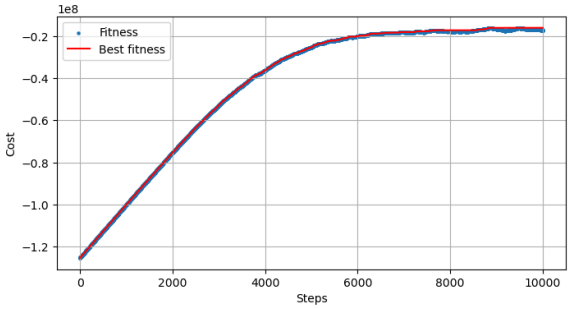
Instance 3 - Simulated Annealing



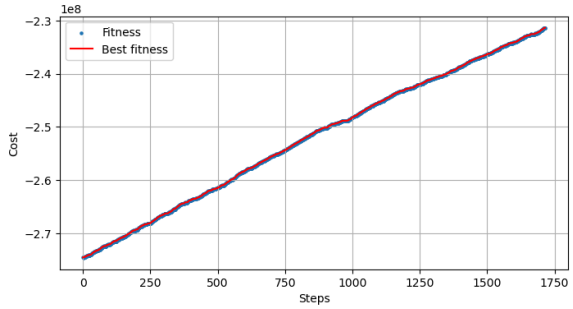
Instance 3 - Tabu Search



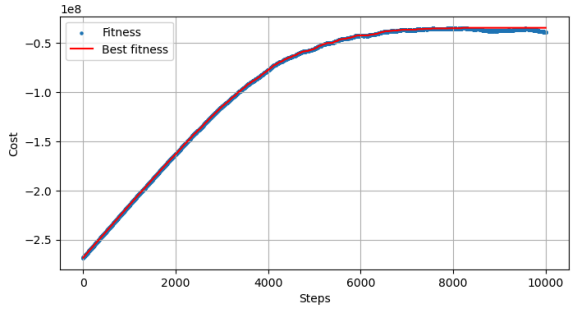
Instance 4 - Simulated Annealing



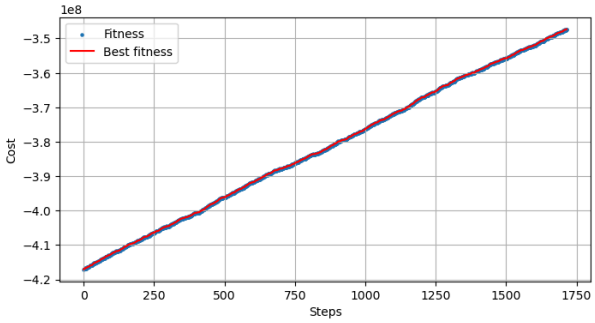
Instance 4 - Tabu Search



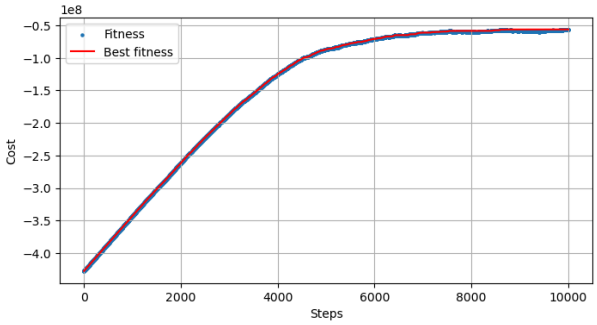
Instance 5 - Simulated Annealing



Instance 5 - Tabu Search



Instance 6 - Simulated Annealing



Instance 6 - Tabu Search

Both **Simulated Annealing (Solution 1)** and **Tabu Search (Solution 2)** exhibit a similar trend when dealing with smaller **Universe Size** and **Num Sets** values. The fitness improves sharply at the beginning but quickly plateaus. However, as these two parameters increase, the fitness improvement becomes more gradual and "smoother." In the case of Simulated Annealing, the improvement curve eventually flattens, turning into a straight line. This means that the best fitness increases almost linearly with larger Universe Size and Num Sets.

Additionally, with smaller Universe Size and Num Sets, both algorithms explore a greater variety of fitness values, sometimes accepting slightly worse solutions temporarily. This is evident from the many scattered blue points, which represent fitness values being tested. As Universe Size and Num Sets increase, the fitness and best fitness plots nearly overlap, indicating that the algorithms quickly converge to a solution and explore fewer fitness values.

Specific Case: Instance 1 (Universe Size: 100, Num Sets: 10)

Both algorithms struggle with instance 1, possibly because there are too few sets. Simulated Annealing quickly finds a maximum and stays stuck there, unable to escape or improve further. On the other hand, Tabu Search starts from a valid solution but fails to find any other valid ones throughout the run. This is reflected by the fact that there are only two blue points (representing valid fitness values), and the best fitness moves directly from the initial to the final one, forming a straight line.

Suggestions from my colleagues

I receive two reviews for my laboratory:

1. https://github.com/GiorgiaModi/CI2024_lab1/issues/1

> I'm actually amazed by how you completed this lab! The code is very well written and organized, with all the needed comments in the right places to make the reader understand better. Also, a complete folder storing the plots indicates attention to details and diligence.

> The README file is clear and explains very well your solutions.
>
> However, if I must find an imprecision in your work I would just say to find a better graph for the 1st instance of the Tabu search algorithm (I can see only 1 step of iteration) and to remove the "-" sign before the cost values in the results table (because the cost is a non-negative quantity talking in real world terms), but the second one is only a formality.
>
> Really a great job overall!!

My answer:

You're absolutely right about the Tabu Search plot for Instance 1—it only found one valid solution, which explains the flat graph. I'll make that more explicit in the documentation.

As for the negative cost values, you're correct—it's more intuitive to show them as positive. I used the negative sign just to reflect maximization in the fitness function, but I'll adjust the table for clarity.

2. https://github.com/GiorgiaModi/CI2024_lab1/issues/2

> I want to start this review by saying that everything in your folder is really well organized. Unlike what I did, you readme file is extremely clean and well structured, making it much easier for the reviewer to grasp everything you've done. Moreover, the table with the summary of all the executions data provides important details that show the crucial differences between the two algorithms: speed vs accuracy. The folder with the plots and the outputs on your notebook show that your solution show that you managed to master both techniques. I liked how you implemented not one but two different solutions in order to tackle the same problem. There is not much else to say really, if I really had to recommend an improvement, it would be to try to implement some optimization techniques for larger datasets (tabu search) since the difference in time with the Simulated Annealing is substantial. However, overall you did a great job, congrats!!

My answer:

I totally agree on Tabu Search's performance for large instances—there's definitely room to optimize. I'm thinking about adding better diversification or smarter neighbor selection. Your suggestion is very helpful and something I'd like to explore further.

Thanks again for taking the time to review my work!

Suggestions for my colleagues

I have made two reviews:

1. https://github.com/Asir29/CI2024_lab1/issues/1

> The code is really well-written and clear, thanks to the helpful comments throughout. The dynamic adjustment of mutation strength is a clever way to boost performance; It might be worth exploring further how to adjust the strength, but I didn't fully understand that part either :). I also liked how you used two separate vectors: one to keep track of all solutions and another just for the valid ones. This way, all solutions appear in the graph, but you still pick the best valid one with the lowest cost.
>
> That said, I'm a bit confused by the mention of Simulated Annealing. In the code, you only accept improvements, whereas from the theory, Simulated Annealing means you sometimes

> accept worse solutions (with p != 0) to avoid getting stuck in local optima. Maybe that part could be clarified? Also, it could be nice to uncomment the code that plots the history, so we can see how the fitness evolves over time.
>
> Overall, though, this is a great piece of work! You've applied a lot of concepts from class in a really clear and simple way. Good job!

2. I didn't copy the review and my colleague delete the repo :(

Laboratory 2

Repository: https://github.com/GiorgiaModi/CI2024_lab2

Traveling Salesman Problem: Given a list of cities and the distances between each pair of cities, what is the shortest possible route that visits each city exactly once and returns to the origin city?

In this lab, I experimented with two different algorithms to solve the Traveling Salesman Problem (TSP): a greedy algorithm and a genetic algorithm.

I implemented features that allow the user to customize the problem setup in the notebook. Specifically, the user can choose the country (i.e., the dataset) by uncommenting the desired line in the second cell, which loads the corresponding CSV file. Moreover, the user can select a specific starting city by specifying it in the third cell.

Solution 1: Greedy algorithm

The greedy algorithm aims to find a suboptimal but efficient solution by selecting the nearest unvisited city at each step. This process continues until all cities are visited, forming a complete cycle. To determine the closest city, I used a distance matrix, which stores the distances between all pairs of cities. While this approach does not guarantee the optimal solution, it provides a fast approximation with minimal computational complexity.

Solution 2: Genetic algorithm

The genetic algorithm, in contrast, seeks an improved solution by simulating natural selection processes. It initializes the population with either random routes or by using slight mutations of a precomputed greedy solution. When no initial solution is provided, the algorithm starts with completely random routes.

In each generation, the algorithm follows a structured process:

- Elitism: The top-performing routes, or "elites," are preserved directly in the new population to retain quality solutions. I chose this approach as it improves solution quality and speeds up convergence.
- Selection and Crossover: The rest of the population is generated by selecting parent routes probabilistically and combining them through crossover. I experimented with three crossover methods:
  - "Classic" Crossover
  - Partially Matched Crossover (PMX)
  - Inver-Over Crossover

  From my experiments, I obtained the best results with the Inver-Over Crossover, which consistently yielded shorter routes and higher-quality solutions.

- Mutation: Each offspring route is subjected to a mutation, introducing small changes to ensure diversity.

Observations

I conducted two main trials with the genetic algorithm:

1. Random Initialization: The initial population was entirely random. I decided to set number of generations equal to 2,500 in this case.
2. Greedy-Based Initialization: The initial population was derived from mutations of the greedy solution. I decided to set number of generations equal to 1,500 in this case.

Both approaches achieved similar final results when the number of cities was low. However, when the number of cities increased, starting from a random solution and running only 2,500 generations did not yield results comparable to those obtained with the greedy algorithm (for example, with the China dataset, the result of the genetic algorithm starting from a random solution yields poor results due to the large size of the dataset). In contrast, beginning with a greedy solution consistently led to improvements in all cases, requiring fewer generations (1,500 vs. 2,500) to reach the optimal route. This demonstrates that starting with a relatively good solution can accelerate convergence, even if the algorithm explores other solutions through crossover and mutation.

Observation: in larger datasets, more than 1500 generations would have been necessary to find an optimal solution even starting from the greedy solution and not from a random one. However, due to time constraints, I set this limit for all cases.

The solutions obtained with the genetic algorithm tend to be closer to the global optimum compared to the greedy algorithm. However, this improved accuracy comes at the cost of significantly longer execution times.

In conclusion, I believe that the best approach is to utilize a genetic algorithm that allows for better results while starting from a greedy solution rather than a random one. This strategy aims to optimize execution times while still providing effective solutions.

Results

Naturally, the results depend on the chosen starting city. In the "plots" folder in the repo (and also reported below), I have included the results obtained for different countries, with starting cities selected at my discretion.

In the table below, I have summarized all the obtained results.

Note again that for large datasets, the results are not optimal and additional generations would have been necessary to achieve better outcomes.

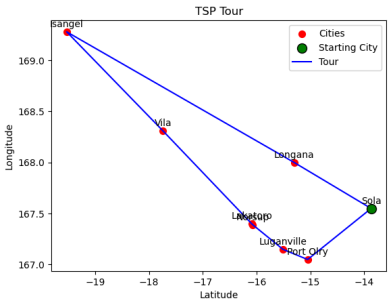
Dataset	Start City	Approach	Final Cost (km)	Number of Steps/Generations	Time
Italy	Trento	Greedy	5172.82	46	0.2s
		Genetic (random start)	4181.63	2500	7m 28.6s
		Genetic (greedy start)	4175.24	1500	4m 31.1s
Vanuatu	Sola	Greedy	1519.35	8	0.1s
		Genetic (random start)	1345.54	2500	1m 41.8s
		Genetic (greedy start)	1345.54	1500	55.2s
US	Tacoma	Greedy	48810.9	326	0.5s
		Genetic (random start)	133710.9	2500	54m 4.4s
		Genetic (greedy start)	47000.24	1500	33m 39.3s
Russia	Armavir	Greedy	43953.66	167	0.3s
		Genetic (random start)	55768.75	2500	26m 44.6s
		Genetic (greedy start)	39870.89	1500	15m 52.8s
China	Guiyang	Greedy	65605.44	725	2.0s
		Genetic (random start)	377135.14	2500	128m 2.8s
		Genetic (greedy start)	64869.18	1500	81m 45.3s

Plots

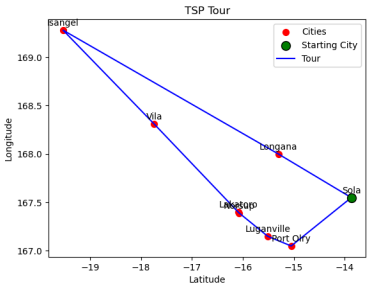
`genetic_greedy_start` represents the result obtained with genetic algorithm starting from a previous computed greedy solution. `genetic_random_start` represents the result obtained with genetic algorithm starting from a random solution. `greedy` represent the greedy solution.

Observation: when the number of cities is too high, plots are not comprehensible.

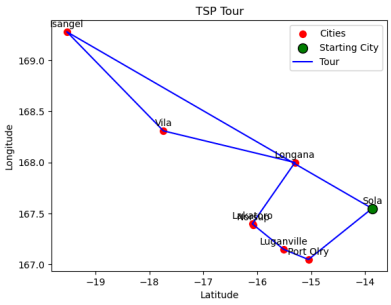
- Vanuatu



vanuatu_genetic_greedy_start

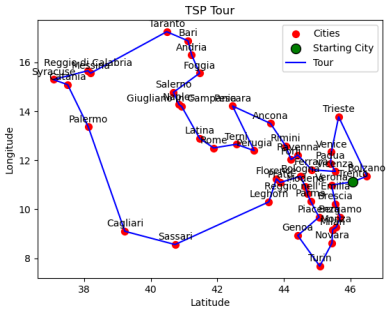


vanuatu_genetic_random_start

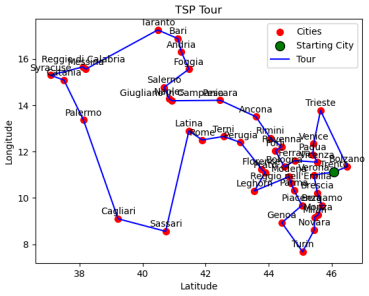


vanuato_greedy

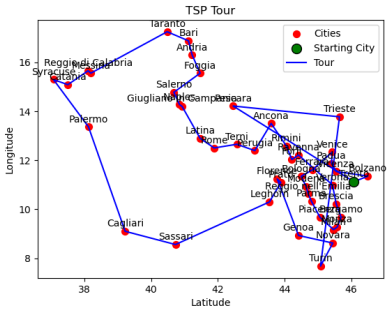
- Italy



italy_genetic_greedy_start

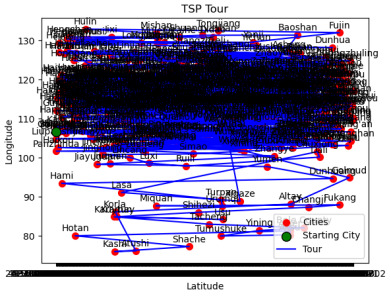


italy_genetic_random_start

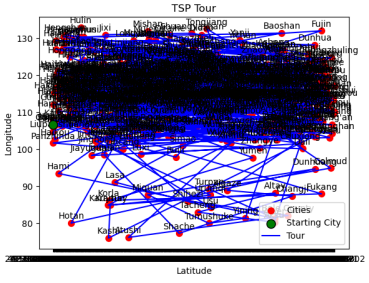


italy_greedy

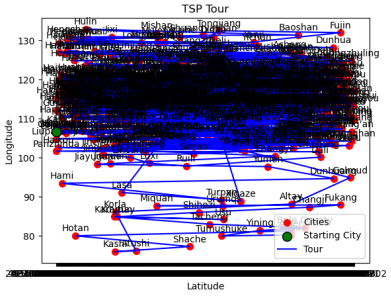
- China



china_genetic_greedy_start

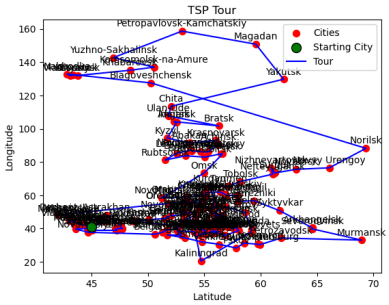


china_genetic_random_start

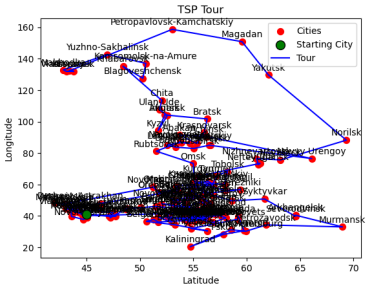


china_greedy

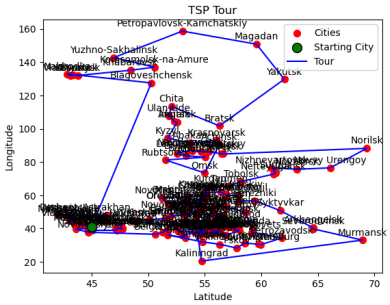
- Russia



russia_genetic_greedy_start

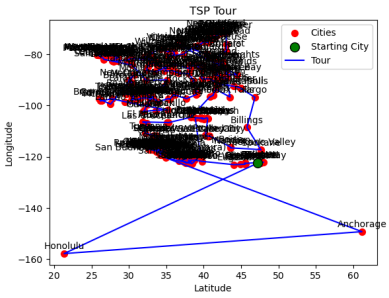


russia_genetic_random_start

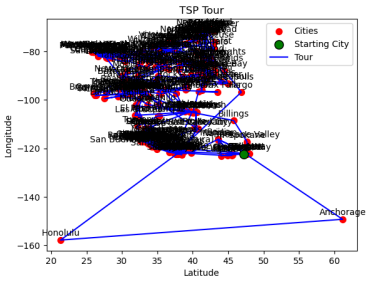


russia_greedy

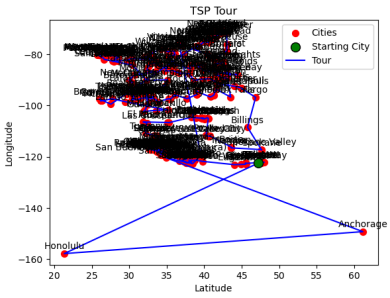
- US



us_genetic_greedy_start



us_genetic_random_start



us_greedy

Suggestions from my colleagues

No one review my Lab2. I sent an email but didn't have any response.

Suggestions for my colleagues

1. https://github.com/LorenzoUgo/CI2024_Lab2/issues/1

> Hi! Regarding the greedy algorithm, I appreciate how you approached differently from the standard method. Instead of starting randomly and moving to the nearest city, you searched for the two closest cities in the first loop to find a more optimal starting point. Even though I imagine this took a bit more time and made the algorithm a bit slower, it clearly paid off, as your greedy results were better than mine.
>
> Regarding the genetic algorithm, I took a look at the results you included in the README. It seems that for all states except Vanuatu, the genetic algorithm didn't perform as well as the greedy solution. One suggestion I have is to initialize the population with the solution from the greedy algorithm instead of using random values. Alternatively, running the algorithm for more generations might help improve the results.
> I also really like how you incorporated various types of mutation, recombination, and parent selection. I like how you vary the mutation type by selecting it at random. For the crossover you chose fixed "inverse_over" and for parent selection "roulette_wheel_fitness_weight." Did you find that these choices led to better performance? Just curious :)

> I noticed that the final plot in the notebook is for a different country than the one you started with, but that’s no big deal.
>
> Overall, good work! Keep it up!

2. https://github.com/aledima00/CI2024_lab2/issues/1

> I've started by looking at your results table, and I see that you’re achieving similar outcomes to mine, but in impressively short runtimes—just 10 minutes for the largest dataset compared to 80 minutes with my algorithm! That’s incredible. Let me start by saying well done; your solution seems more reproducible too, which is a big plus.
>
> Next, I went through your code, and I have to say it's excellently written. It’s very clear and easy to follow, thanks to the numerous comments and explanations before each section. This makes it much easier to understand the flow and purpose of each part—great job on that!
>
> As for the greedy algorithm, I like that you modified the “standard” approach a bit by building the solution with an alternating strategy—choosing the closest city to either the start or the end of the current path. I’m not entirely sure if this gives better results than the typical greedy approach, which always moves to the closest city, but it would’ve been interesting to see a comparison with standard greedy results in the README table. It would have let me directly compare your approach with mine. Either way, good call on trying out something different!
>
> Regarding the genetic algorithm, you've done an incredibly clean and precise job. I noticed that the algorithm cycles through three mutation operators and three crossover operators, applying them sequentially one by one. I like the variety here, but I'm curious why you chose to switch them every time rather than just sticking with one mutation and one crossover operator. (Not a criticism, just a question!)
>
> As for the parameters that are linearly dependent on POPULATION_SIZE (which itself is tied to GENOME_SIZE), I'm really interested in the formulas you used—they definitely make sense (much better than fixed values!), even though I don’t fully understand them.
>
> The Exploration Boost feature is super interesting too, allowing the algorithm to slow down when there’s no improvement after a certain number of iterations is a clever approach.
>
> Amazing work! The most impressive thing, I have to say again, is the great results achieved in such a short execution time. I’m sure it’s thanks to all these techniques you experimented with. Really well done!

Laboratory 3

Repository: https://github.com/GiorgiaModi/CI2024_lab3

This report explains two different algorithms to solve the **n^2-1 puzzle**. The puzzle consists of a square grid (with a dimension of n x n) with tiles numbered from 1 to n^2-1, and an empty space (0) that allows the tiles to slide.

The goal of the puzzle is to arrange the tiles in a particular order by sliding them into the empty space. The challenge is to find the sequence of moves that leads from a scrambled initial configuration to the goal configuration, using the fewest number of moves.

For example, for a 4×4 puzzle (n = 4), the goal configuration is:

```
1  2  3  4
5  6  7  8
9  10 11 12
13 14 15 0
```

Solution Strategies Implemented

Two search algorithms were implemented to solve the puzzle: **A* Search** and **Depth-First Search with Bound (Iterative Deepening DFS)**. Both algorithms find the sequence of moves needed to transition from an initial scrambled state to the goal state.

Algorithm 1: A* Search

A* is a popular **informed search algorithm** that uses both the current cost (g-score) and a heuristic (h-score) to prioritize the most promising states to explore.

- **g-score**: The cost of reaching the current state from the start.
- **h-score**: A heuristic estimate of the cost from the current state to the goal. In this case, the heuristic used is the **Manhattan Distance**, which calculates the sum of the vertical and horizontal distances between the current position and the target position of each tile.
- **f-score**: The sum of the g-score and h-score (`f = g + h`), which is used to prioritize which states to explore first. States with the lowest f-score are expanded first.

Steps:

1. Start from the initial scrambled state.
2. Use a priority queue (min-heap) to store states, where the priority is determined by the f-score.
3. Expand the state with the lowest f-score, calculate its neighbors, and add them to the queue.
4. Repeat the process until the goal state is found or no solution is possible.

A* Search garantees the optimal solution with the least number of moves, but it's more memory-intensive due to storing all explored states and their f-scores.

Algorithm 2: Depth-First Search with Bound (Iterative Deepening DFS)

Depth-First Search (DFS) is an uninformed search algorithm that explores the search space by going as deep as possible down one path before backtracking. The algorithm with a bound places a limit on the depth of the search to prevent it from going infinitely deep.

Iterative Deepening DFS is an enhancement of DFS that gradually increases the depth limit, performing DFS for each depth limit starting from 0 until a solution is found or the maximum depth is reached.

Steps:

1. Start from the initial scrambled state.
2. Perform DFS within the first depth limit.

3. If the solution is not found, increment the depth limit and repeat the process.
4. Continue until the goal state is reached or the maximum depth is reached.

Depth-First Search with Bound is more memory-efficient but can take longer to find a solution compared to A*.

Results

Execution time increases significantly as the size of the puzzle increases. For sizes larger than 3×3, the execution time is highly variable. For example for 4×4, the time can range from 1-2 minutes to up to 30 minutes, while for 5×5, it can take more than two hours.

Execution time significantly depends on the number of RANDOMIZED_STEPS used to generate the initial configuration. For 5×5 puzzles and larger, reducing RANDOMIZED_STEPS to 100 yields a solution in a reasonable amount of time. However, even adding just a small amount to RANDOMIZED_STEPS results in a dramatic increase in execution time, making larger puzzles infeasible. Additionally, for DFS, increasing the `max_depth` to 25 has been tested to better handle the complexity of these puzzles.

A new heuristic function, **linear conflict**, was also implemented but did not improve performance compared to the Manhattan distance heuristic.

In general, the results are **very** dependent on the initial configuration, so the results shown in this table are only indicative. For DFS with bounds, it can happen that no solution is found within the given depth limit, so the depth limit should be increased accordingly.

Method	Size	Time	Initial Step	Final Step	Notes
A*	2×2	0.01s	`[[3 1]` `[2 0]]`	`[[1 2]` `[3 0]]`	RANDOMIZED_STEPS = 10,000, #steps = 4

	3×3	0.01s	`[[4 1 6]`
			`[3 0 5]`
			`[2 7 8]]`
				`[[1 2 3]`
				`[4 5 6]`
				`[7 8 0]]`
					RANDOMIZED_STEPS = 10,000, #steps = 18

	4×4	1m 38.6s	`[[13 14 7 6]`
			`[3 0 1 8]`
			`[2 12 9 15`
			`[11 10 4 5]]`
				`[[1 2 3 4]`
				`[5 6 7 8]`
				`[9 10 11 12]`
				`[13 14 15 0]]`
					RANDOMIZED_STEPS = 10,000, #steps = 52

	5×5	1.5s	`[[7 3 4 5 10]`
			`[2 0 1 8 9]`
			`[6 12 13 19 14]`
			`[11 16 18 24 15]`
			`[21 17 22 23 20]]`
				`[[1 2 3 4 5]`
				`[6 7 8 9 10]`
				`[11 12 13 14 15]`
				`[16 17 18 19 20]`
				`[21 22 23 24 0]]`
					RANDOMIZED_STEPS = 100, #steps = 24

	6×6	1.5s	`[[1 15 2 10 11 6]`
			`[3 0 4 5 9 12]`
			`[7 8 14 16 17 18]`
			`[13 20 21 28 22 24]`
			`[19 26 33 27 23 30]`
			`[25 31 32 34 29 35]]`
				`[[1 2 3 4 5 6]`
				`[7 8 9 10 11 12]`
				`[13 14 15 16 17 18]`
				`[19 20 21 22 23 24]`
				`[25 26 27 28 29 30]`
				`[31 32 33 34 35 0]]`
					RANDOMIZED_STEPS = 100, #steps = 36

	7×7	0.2s	`[[ 1 2 3 4 5 6 7]` `[ 8 9 10 11 12 13 14]`	`[[ 1 2 3 4 5 6 7]` `[ 8 9 10 11 12 13 14]`	RANDOMIZED_STEPS = 100, #steps = 36

Method	Size	Time	Initial Step	Final Step	Notes
			[15 16 18 25 26 19 21] [23 29 17 39 33 20 28] [0 30 31 24 32 27 34] [22 45 44 38 40 41 35] [37 36 43 46 47 48 42]]	[15 16 17 18 19 20 21] [22 23 24 25 26 27 28] [29 30 31 32 33 34 35] [36 37 38 39 40 41 42] [43 44 45 46 47 48 0]]	
DFS	2×2	0.01s	[[3 1] [2 0]]	[[1 2] [3 0]]	RANDOMIZED_STEPS = 10,000, Depth: 4, #steps = 4
	3×3	0.01s	[[4 1 6] [3 0 5] [2 7 8]]	[[1 2 3] [4 5 6] [7 8 0]]	RANDOMIZED_STEPS = 10,000, Depth: 20,#steps = 20
	4×4	4.1s	[[13 14 7 6] [3 0 1 8] [2 12 9 15] [11 10 4 5]]	-	RANDOMIZED_STEPS = 10,000, No solution found (Depth limit: 20)
	5×5	1m 13.5s	[[7 3 4 5 10] [2 0 1 8 9] [6 12 13 19 14] [11 16 18 24 15] [21 17 22 23 20]]	[[1 2 3 4 5] [6 7 8 9 10] [11 12 13 14 15] [16 17 18 19 20] [21 22 23 24 0]]	RANDOMIZED_STEPS = 100, Depth: 24,#steps = 24
	6×6	11m 37.7s	[[1 15 2 10 11 6] [3 0 4 5 9 12] [7 8 14 16 17 18] [13 20 21 28 22 24] [19 26 33 27 23 30] [25 31 32 34 29 35]]	-	RANDOMIZED_STEPS = 100, No solution found (Depth limit: 25)
	7×7	31m 47.3s	[[1 2 3 4 5 6 7] [8 9 10 11 12 13 14] [15 16 18 25 26 19 21] [23 29 17 39 33 20 28] [0 30 31 24 32 27 34]	-	RANDOMIZED_STEPS = 100, No solution found (Depth limit: 25)

Method	Size	Time	Initial Step	Final Step	Notes

			`[22 45 44 38 40 41 35]`
			
			`[37 36 43 46 47 48 42]]`		

`RANDOMIZED_STEPS` refers to the number of random moves (or randomization steps) used to generate the initial configuration of the puzzle before the algorithm begins solving it.

`#steps` refers to the number of steps (i.e., individual moves or edits) taken by the algorithm to reach the goal solution, where each step represents a change in the matrix configuration.

Suggestions from my colleagues

1. https://github.com/GiorgiaModi/CI2024_lab3/issues/2

> Hi Giorgia, great work on your implementation!
>
> Your explanation of the project and the design choices you made are really impressive, it's detailed, well-structured and you've clearly put a lot of thought into explaining the algorithms and presenting the results.
>
> I'd like to highlight some of the excellent aspects of your work and also discuss a few areas where there might be opportunities for optimization.
>
> You introduced the $n^2 - 1$ puzzle very well providing a clear description of the rules and the goal.
>
> Including the example of the 4×4 goal configuration makes it easy for readers to visualize the problem.
>
> This is especially helpful for anyone unfamiliar with the puzzle.
>
> The choice of A* and Iterative Deepening Depth-First Search (DFS) shows a solid understanding of the trade-offs between informed and uninformed search algorithms.
>
> Explaining the strengths and weaknesses of each algorithm—such as A* being optimal but memory-intensive and DFS being memory-efficient but slower—is a strong point in your explanation.
>
> Using the Manhattan Distance for A* was an excellent decision: this heuristic is admissible and consistent making it well-suited for the puzzle since movements are restricted to vertical and horizontal directions.
>
> It's clear you considered the problem's constraints carefully when choosing the heuristic.
>
> Implementing a second heuristic, linear conflict, demonstrates creativity and a willingness to experiment with different approaches. Even though it didn't improve performance, testing it shows your analytical approach to problem-solving.
>
> The results table is very well-organized and informative because it provides valuable insights into how the algorithms perform with different puzzle sizes and configurations.

> Mentioning how the performance depends on factors like `RANDOMIZED_STEPS` and `max_depth` is a great touch.
>
> It highlights the variability of the problem and gives readers context for interpreting the results.
>
> In conclusion, you've shown a deep understanding of the algorithms by explaining how they handle larger puzzles and discussing why certain configurations might fail (for example, DFS not finding solutions within the depth limit).
>
> Your explanation of DFS with bounds is clear but the results show that it struggles with larger puzzles due to the depth limit. It might be worth exploring alternative uninformed search algorithms, such as Breadth-First Search (BFS), for smaller puzzles, or implementing a more adaptive depth limit strategy.
>
> While the results table is detailed, it could benefit from more analysis. For example:
>
> - Why does DFS take significantly longer for larger puzzles compared to A*?
> - Why does reducing `RANDOMIZED_STEPS` improve execution time for larger puzzles?
> - How does the initial configuration impact the performance of each algorithm?
>
> Adding this context would make the results even more insightful.
>
> Although the linear conflict heuristic didn't outperform Manhattan Distance, it would be great to include a short explanation of why this might be the case. For instance:
>
> - Linear conflict might add computational overhead without significantly improving the heuristic's informativeness.
> - Highlighting such insights would strengthen your analysis and demonstrate your understanding of heuristic design.
>
> Overall, Giorgia, your implementation is impressive and well-documented. You've clearly put a lot of effort into explaining the algorithms, their trade-offs and the results.
>
> Great work overall—keep it up! 😊

I have tried to answer to the questions of my colleague:

- **Why does DFS take significantly longer for larger puzzles compared to A^*?** Because DFS doesn't know which path is better—it just goes deep one way and only backtracks when stuck. So for big puzzles, it wastes time exploring many bad paths. A^* is smarter: it uses a heuristic to guess which moves bring it closer to the goal.
- **Why does reducing `RANDOMIZED_STEPS` improve execution time for larger puzzles?** If the puzzle is scrambled less (fewer random steps), it's already closer to the solution. So the algorithm finds the goal faster. More random steps = more complex start = longer solving time.
- **How does the initial configuration impact the performance of each algorithm?** The harder the starting position, the more steps and time are needed. For both A^* and DFS, some positions are just easier to solve. So even with the same algorithm, performance can change a lot depending on how scrambled the puzzle is.
- **Why didn't the linear conflict heuristic outperform Manhattan Distance?** Because even if linear conflict is more detailed, it takes more time to compute and didn't give much better results. Manhattan is simple, fast, and works well—so in practice, it was better overall.

2. https://github.com/GiorgiaModi/CI2024_lab3/issues/1

> Hi there, I'm Andrea. First of all, you did a great job in this lab!
>
> I found the README file really useful to understand the algorithms you implemented, as well as the tables containing the results in terms of execution time and performance.
>
> **Comments & suggestions**
>
> I have a few suggestions to improve your notebook.
>
> 1. I saw that in some cases your DFS algorithm doesn't find a solution. In these cases, I think it should be worth trying to increase the depth. E.g. for the 4×4 puzzle with 10k randomization timesteps and a maximum depth of 20 your algorithm fails to find a valid solution.
> 2. You might also consider implement another famous algorithm: Breadth First Search (BFS), which is similar to the DFS you implemented. You just need to pop items from the front of the queue, and this can be performed by changing from `stack.pop()` to `stack.popleft()` in the DFS implementation (in this case a better suiting name might be `queue`, instead of `stack`)
>
>    ```
>    # ...
>    while stack:
>        current_state, path, depth = stack.popleft()
>    # ...
>    ```
>
> 3. Include some plots on how the metrics changes during the problem solving, in order to check the convergence speed towards the solution (and also to visualize if the DFS algorithm stops near the solution or not)

My answer:

- **DFS depth issue**: You're totally right — in cases like the 4×4 puzzle with many random moves, the depth limit of 20 was too low. Increasing it helps, but it also slows things down a lot. I'll try tuning it better or making the depth limit adaptive.
- **Breadth-First Search (BFS)**: Great idea! Since BFS is similar to DFS but guarantees the shortest solution for uninformed search, I'll consider adding it. Using `popleft()` instead of `pop()` is a simple change but makes a big difference. Thanks for the code tip!
- **Plots and metrics**: I didn't include visualizations yet, but I agree that plotting things like f-score over time or number of explored states would help understand how the search progresses (especially to see if DFS stops close to the goal). I'll try adding some simple plots next.

Suggestions for my colleagues

1. https://github.com/Peppe2212/CI2024_lab3/issues/1

> First of all, great job! I agree with the use of A* to solve the problem—I used the same approach as well. The considerations you made in the README about DFS are likely correct (the branching factor is quite high, so DFS risks running out of memory quickly). Since you pointed out that choosing the right heuristic function is crucial for A*, you could have tried

> experimenting with more than one heuristic and compared the results (just a suggestion, not a criticism!).
>
> Regarding the implementation of the algorithm, the code is well-written and highly readable. However, you could have tested other puzzle configurations, like 4×4 or 5×5, to see if the algorithm performs equally well, as I've noticed that challenges often arise with larger dimensions.
>
> These are just my suggestions, but overall, excellent work! Keep it up!

2. https://github.com/LucianaColella7/CI2024_lab3/issues/1

> First of all, great job! I also decided to use A* as the algorithm and faced similar issues when the puzzle size increased. Like you, I had to reduce the number of randomizing steps to generate solvable puzzles within reasonable time limits. Maybe you can also try solving even larger configurations (like 6×6, 7×7, etc.) to see how your implementation performs on those.
>
> I really appreciate that you experimented with multiple heuristics, as it's true that a particular heuristic might not be suitable for certain types of problems. I found it interesting how you justified your choice of heuristic in the section "manhattan_distance vs. combined_heuristics."
>
> Another thing I found very helpful (and something I didn't do myself but would have been quite useful) is how you reported both the length of the solution path (i.e., Quality - number of actions in the solution) and the number of steps the algorithm took to find that solution (i.e., Cost - total number of actions evaluated). This approach makes it easier to justify longer execution times in a clear way.
>
> Lastly, precomputing the goal configuration and using it for the Manhattan distance is a great idea to save execution time!
>
> Overall, I don't have much to suggest—you've done an excellent job. The code is well-organized and readable, and the README is clear and informative.
>
> Awesome work! Keep it up!

Project report - CI2024 Project – Symbolic Regression

This project was entirely developed in collaboration with my colleague Andrea Delli (s331998).

All code was primarily developed within my colleague's GitHub repository (https://github.com/RonPlusSign/CI2024_project-work) and later cloned in the repository on my account (https://github.com/GiorgiaModi/CI2024_project-work) upon completion.

Problem Statement

Symbolic regression is the task of discovering a mathematical expression that best fits a given dataset without assuming any fixed parametric form in advance.
Given samples of inputs
$x \in \mathbb{R}^n$ and outputs $y \in \mathbb{R}$, our goal is to automatically find an analytic function $f(x)$ such that $f(x_i) \approx y_i$ with minimal error.

Methodology

1. **Representation of Individuals**

   Each candidate function is an expression and is represented as a tree of nodes. Each node can be:

   - **Primitives**: operators/functions such as **add**, **sub**, **mul**, **div**, **pow**, **sin**, **cos**, **exp**, **log**, **abs**, etc. These are intermediate nodes of the tree.
   - **Terminals**: input variables $x[0], \ldots, x[n-1]$, constants drawn from a uniform pool in $[-50, 50]$, special constants (π, e), integers, and powers of 10. These are leaf nodes of the tree.
     Note: during developing, we tried different ranges of constant pools.

2. **Initialization**

   We generate an initial population of size P by sampling random trees up to a **maximum depth** D.

   It's possible to restart form a previous saved solution by providing the correct path of the `.pkl` file (using the `solution_path` argument). In this way the genetic algorithm can be interrupted and restored in multiple runs.

3. **Fitness Evaluation**

   As fitness function we use the **Mean Squared Error (MSE)** between predicted $\hat{y}$ and true y:

   $$\text{MSE} = \frac{1}{n} \sum_{i=1}^{n} (y_i - \hat{y}_i)^2$$

   Invalid expressions (shape mismatch, NaNs, exceptions) are assigned infinite fitness.

4. **Parent Selection**

   We experimented with multiple strategies of parent selection:

   - **Tournament Selection:** This method selects a random subset of the population (default size of 10) and chooses the two individuals with the best fitness (lowest error) from within it. This introduces strong selective pressure, promoting rapid convergence toward good solutions. The higher the tournament size, the more likely it is that strong individuals dominate the population early.
   - **Rank Selection**: individuals are ranked by fitness, and selection probabilities are assigned based on their rank, with better-ranked individuals having a higher chance of being selected. This softens the pressure compared to tournament selection and helps maintain genetic diversity by giving lower-performing individuals a non-zero chance of selection.
   - **Fitness-Proportional (Roulette Wheel)**: Also known as "roulette wheel" selection, this approach assigns selection probabilities proportional to the inverse of the fitness score ($1/(\text{fitness} + \varepsilon)$). This rewards individuals with lower mean squared error (MSE) by giving them a higher probability of being selected, but still allows for occasional sampling of less fit individuals. The small ε term ensures numerical stability and prevents division by zero.

5. **Genetic Operators**

   The following mechanisms are implemented to maintain genetic diversity while guiding the search toward better solutions:

   - **Crossover**: it selects a random subtree from each parent and swaps them. To maintain structural constraints, the algorithm retries the swap if the resulting offspring exceed the allowed maximum tree depth.
   - **Mutation**: it introduces variability by randomly altering parts of the individual. Two types of mutation are used, selected uniformly:
     - **Subtree Mutation**: replaces a randomly chosen subtree with a newly generated random subtree. The probability of performing the mutation increases with the depth of the node, ensuring deeper parts of the tree are more likely to be modified—thus increasing structural diversity where it matters most.
     - **Point Mutation**: alters a single node by either changing an operator (while preserving its arity), replacing a variable, or tweaking a constant. Like subtree mutation, the probability increases with depth, allowing fine-grained adjustments to evolve expressions incrementally.
   - **Elitism**: the top $E\%$ of individuals (by fitness) are carried unchanged into the next generation to preserve the best solutions.

   Each genetic operation ensures that the maximum depth constraint is respected.

6. **Termination Criteria**

   The algorithm runs for a fixed number of generations G or until a solution reaches a predefined MSE threshold given as argument (early stopping).

   At the end of the generations, the population is saved to disk (pickle) to allow warm-starting on future runs.

Experimental Setup

To evaluate our symbolic regression framework, we applied it independently to each dataset in the form `problem_i.npz`. For each dataset, we configured three key hyperparameters:

- **Population Size** (P_i): number of candidate solutions per generation.
- **Number of Generations** (G_i): maximum number of evolutionary cycles.
- **Maximum Tree Depth** (D_i): depth limit for generated expression trees to control model complexity and overfitting.

These hyperparameters were **empirically tuned** for each problem to strike a balance between computational cost and the quality of the resulting expressions.

Results

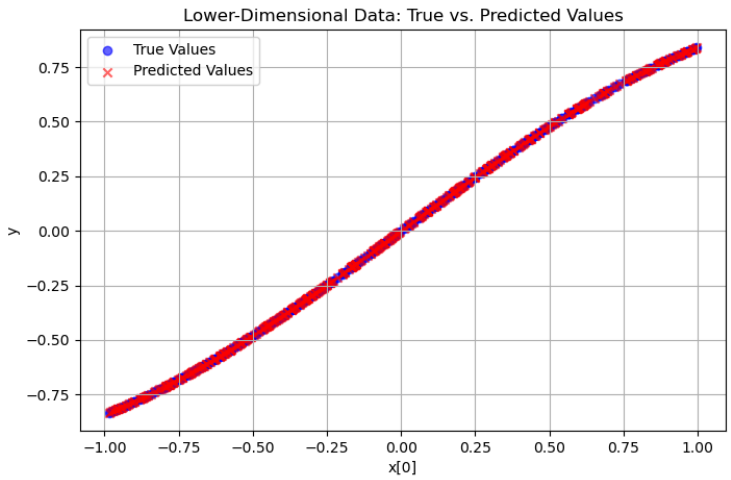
Below are the final symbolic expressions generated for each target function, along with their corresponding Mean Squared Error (MSE).

To better visualize model performance, we also include 3D scatter plots comparing the true vs. predicted values. Each plot represents the output distribution over pairs of input dimensions (e.g., $x[1]$ vs. $x[2]$, $x[1]$ vs. $x[3]$, etc.).

Function f1

- **MSE:** 0
- **Expression:**

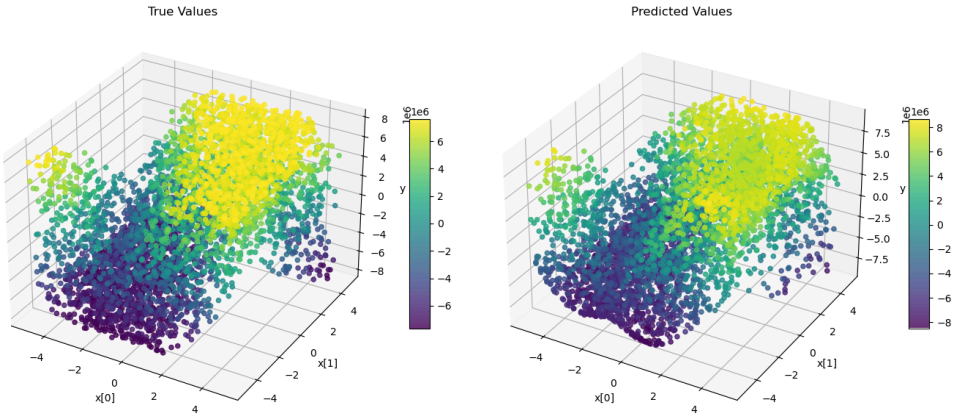
```
np.sin(x[0])
```

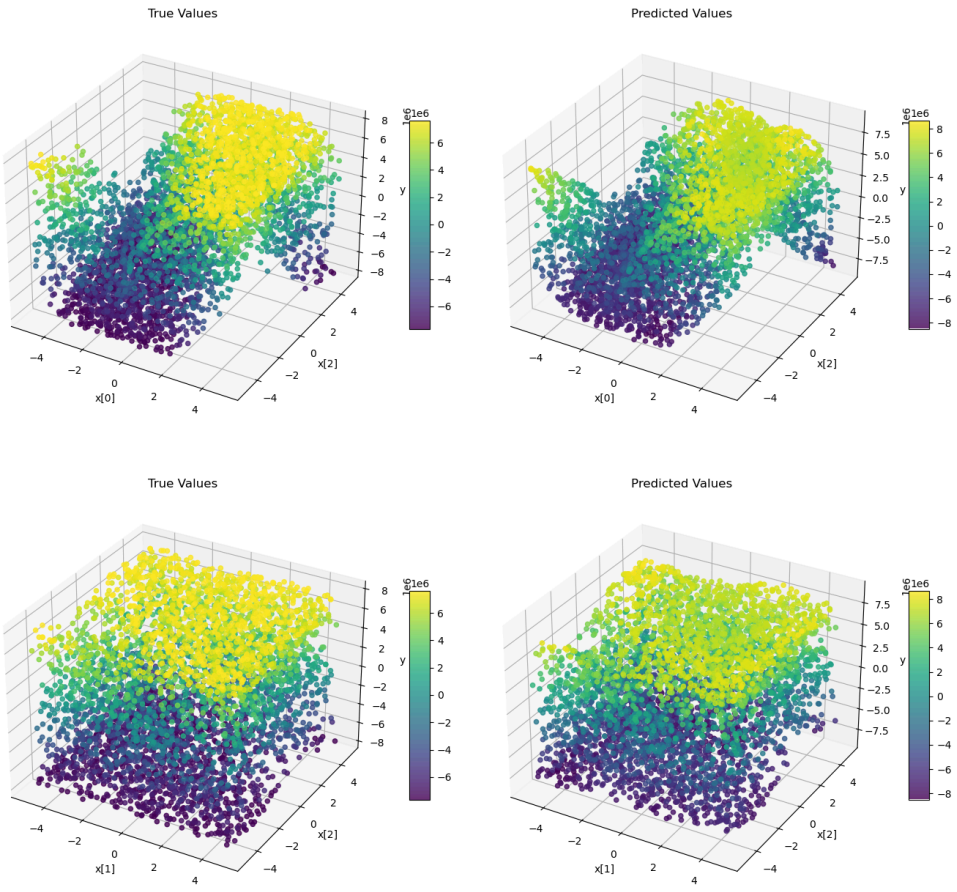


Function f2

- **MSE:** 2.241×10^{12}
- **Expression:**

```
(((((17.976 + (13.556 + 98.991)) + ((34.785 + np.cos(x[2])) - (x[0] * (x[0] + x[0])))) * (((7.68 + np.cos(x[2])) + (x[2] + 98.991)) + (((x[0] * x[2]) - (x[0] * x[1])) - (x[0] * (x[1] + x[0])))))) * (((((x[1] * x[2]) - np.exp(x[2])) + (98.991 + 84.959)) + ((90.002 / x[0]) * (x[1] + x[2])))) + ((((11.47 * x[2]) + (x[1] + x[0])) + ((x[1] + x[0]) + (x[1] + x[0])))) * (((x[0] + x[0]) / (x[0] * 5)) - (x[1] + x[0])))))) * ((np.tanh(((x[0] - (x[0] - x[1])) + 6)) * x[0]) / (((np.exp(np.sin(x[2])) + np.abs((x[1] + x[0])))) + (((x[0] - x[1]) + np.exp(x[1])) - np.tanh((x[0] / x[2]))))) ** ((((x[2] + x[2]) + (x[1] + x[0])) * x[2]) / (((x[2] - 34.785) + (x[2] + 350)) + (np.abs(x[0]) * (x[0] * x[1])))))))
```

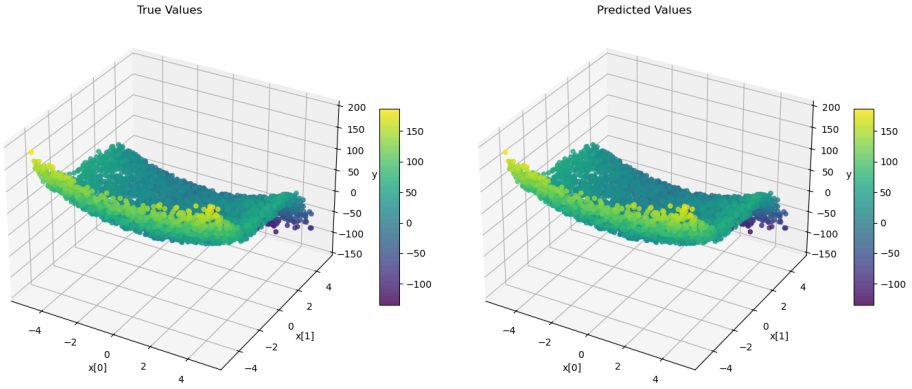


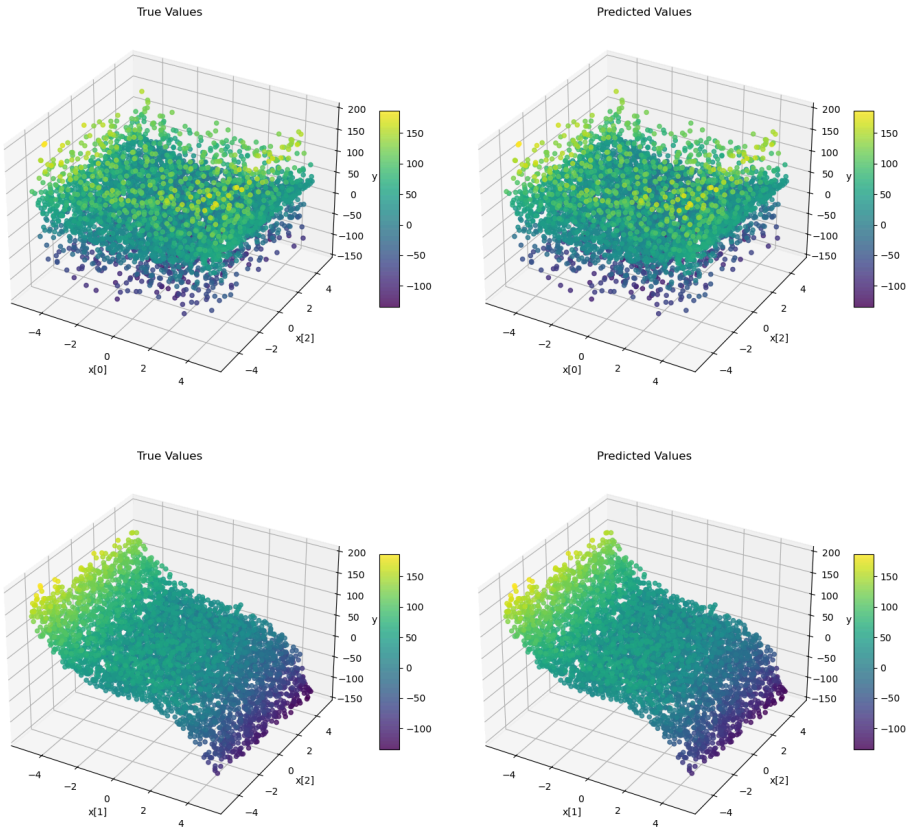


Function f3

- **MSE:** 3.31370×10^{-29}
- **Expression:**

```
((((x[0] * x[0]) + (x[0] * x[0])) + 4) - (((x[2] * 14) / 4) + (x[1] * (x[1] * x[1]))))
```

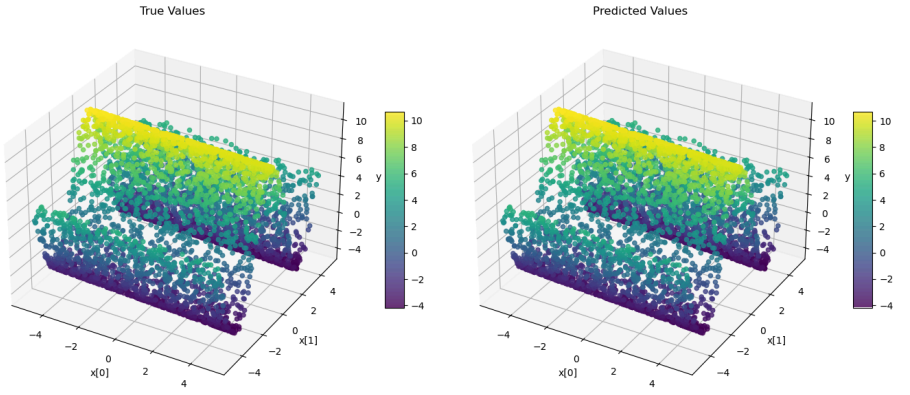




Function f4

- **MSE:** 2.13754×10^{-05}
- **Expression:**

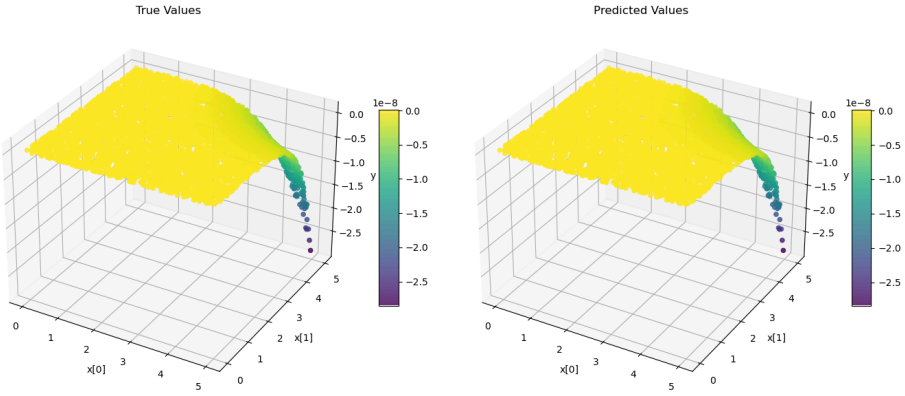
```
(np.tan((((x[0] / -7.911) * np.cos(-13.373)) - np.tan(np.exp(-5.089)))) + (((np.cos(x[1]) + np.cos(-13.44)) * 7) + np.tan(-4.021)))
```



Function f5

- **MSE:** 1.91664×10^{-22}
- **Expression:**

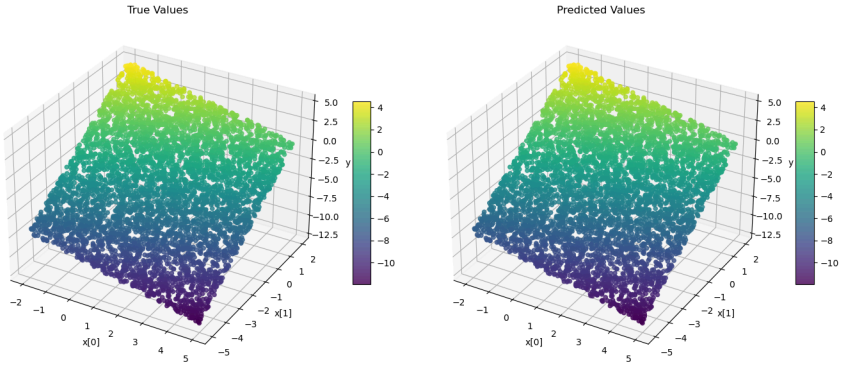
2.8520706810421615e-10 * ((((x[0] ** x[1]) - ((((np.tan(-39.724) - np.cos((-20.187 + np.log(x[0])))) / 4.38) - np.log(x[1])) - (-18.065 - np.log(np.abs(np.abs((np.sin(-18.339) * np.sin(x[1])))))))) / (4.484 + (np.abs(np.log(19.817)) + -36.106))))



Function f6

- **MSE:** 5.73586×10^{-5}
- **Expression:**

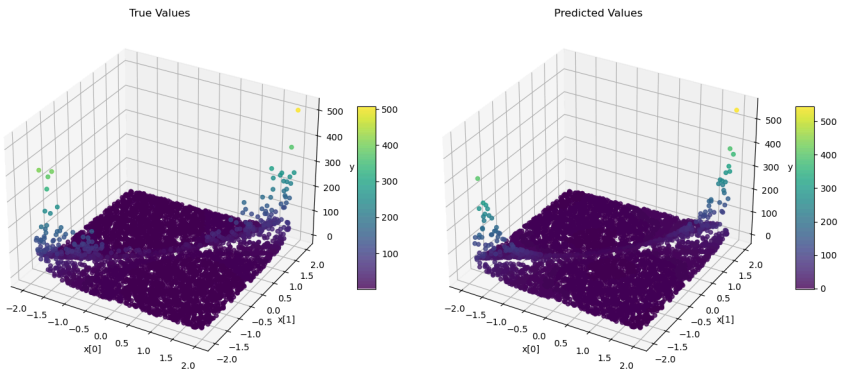
((x[1] + x[1]) - (x[0] + ((x[0] / (np.cos(np.tan(-3.547)) - np.tan(np.abs(-20.187)))) - (x[1] / (np.cos((-12.357 + np.abs(np.tan(20.446)))) - np.tan(np.abs(np.tan(20.446))))))))



Function f7

- **MSE:** 5.02807×10^{1}
- **Expression:**

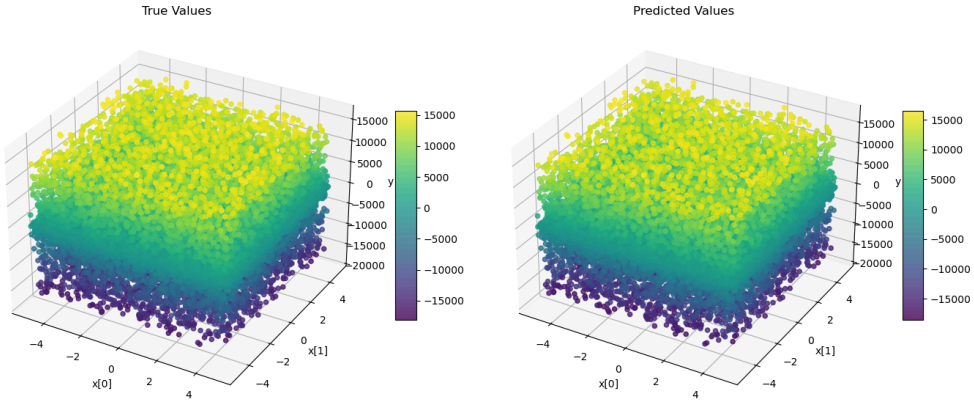
(((((x[1] + ((10 / 92.147) + np.sin(x[1]))) + (x[1] / (np.tanh(x[0]) / x[1]))) + (np.sin((x[1] - (x[0] - x[1]))) + (np.sin(x[0]) + (np.sin(x[0]) + (x[1] + x[1]))))) * (x[0] / np.cos(np.cos((x[1] - x[0]))))) + np.exp(((2 + ((x[0] * x[1]) + ((9 - x[0]) / (20.038 + x[0])))) * np.cos(((200 / (24.951 - 60.469)) * np.tanh((x[0] - x[1])))))))

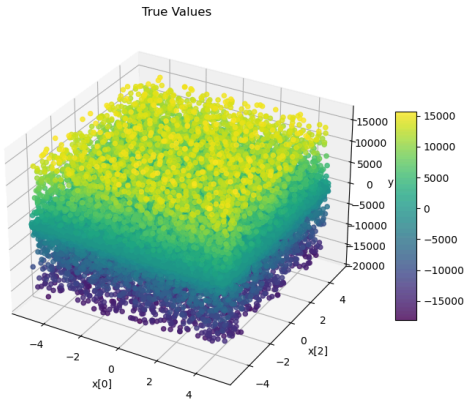


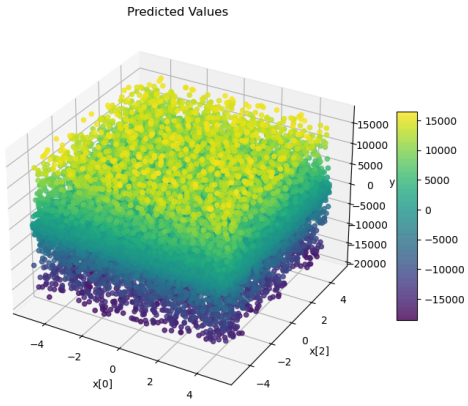
Function f8

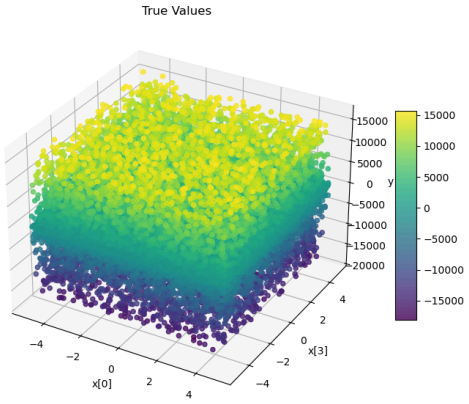
- **MSE:** 2.33539×10^4
- **Expression:**

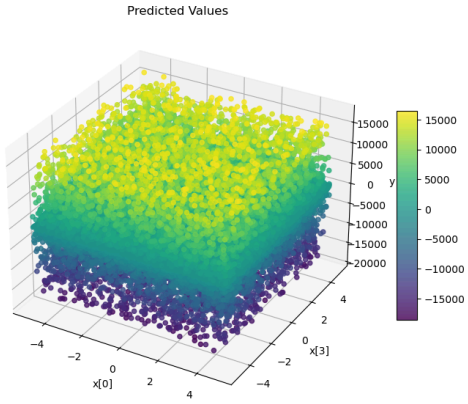
```
((np.exp(x[3]) - (((36.419 - (((-48.79 + -40.34) - (((33.294 + (36.419 - x[5])) / np.exp(np.cos(x[5]))) ** np.cos(x[4]))) * (np.cos(np.sin(np.exp((x[4] + -40.34)))) - (np.sin((x[4] + (-11.033 + x[4]))) - (np.abs(np.cos((x[5] - 27.598))) + np.sin(x[5])))))) + ((((x[4] + np.sin(x[3])) - ((np.sin((np.exp(-1.663) - (x[4] + x[4]))) - ((x[4] + x[4]) + x[4])) * (-1.663 - (x[4] + (x[4] + x[4])))))) + (np.exp(np.tan(np.sin(x[3]))) - ((x[3] - 41.92) * (np.cos(x[4]) - x[3])))) - ((np.sin((np.tan(np.cos(-1.663)) - (x[4] + x[4]))) - (x[4] + (np.cos(np.log(26.12)) + (x[4] + x[4])))) * ((np.sin((np.tan(np.cos(-1.663)) - (x[4] + x[4]))) - (x[4] + np.sin(43.067))) - x[4])))) - ((-48.79 * np.sin(x[5])) + x[5]))) + (np.abs(-21.878) * ((((np.abs(np.abs(33.294)) - ((np.exp(-1.663) - (x[4] + x[4])) * (np.cos(-1.663) - x[4]))) + -22.214) * np.exp(np.exp((np.exp((np.abs(-16.989) - ((np.sin(x[4]) - (x[4] + x[4])) * ((41.697 ** -11.033) - (x[4] + x[4])))))) * -26.183)))) + (np.exp(np.abs(x[5])) * x[5]))))
```

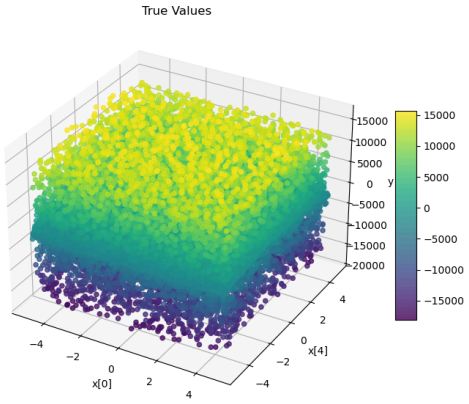


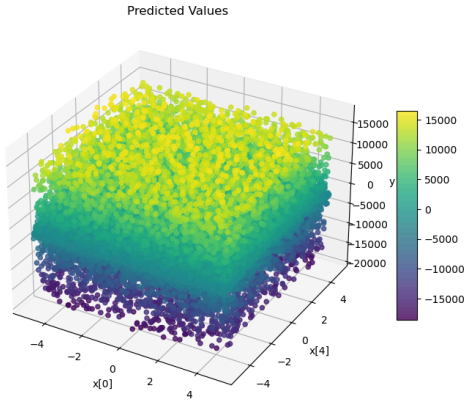

True Values
x[0]
x[2]
y

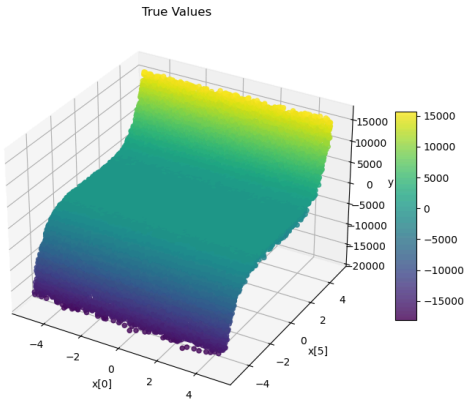

Predicted Values
x[0]
x[2]
y

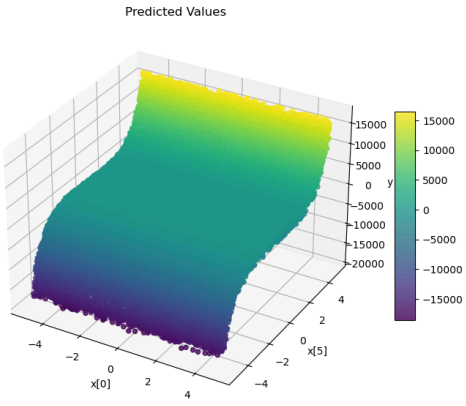

True Values
x[0]
x[3]
y

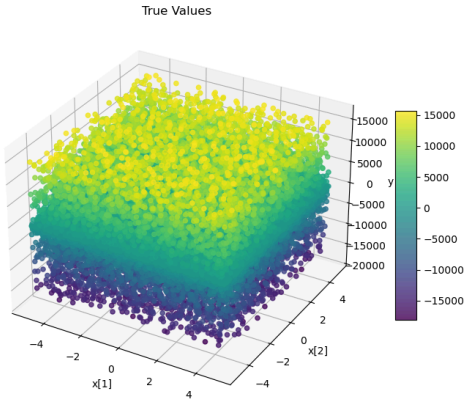

Predicted Values
x[0]
x[3]
y

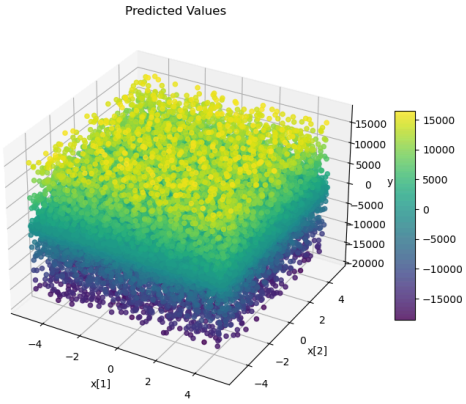

True Values
x[0]
x[4]
y

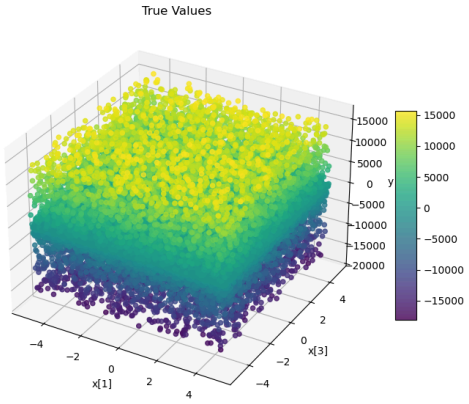

Predicted Values
x[0]
x[4]
y

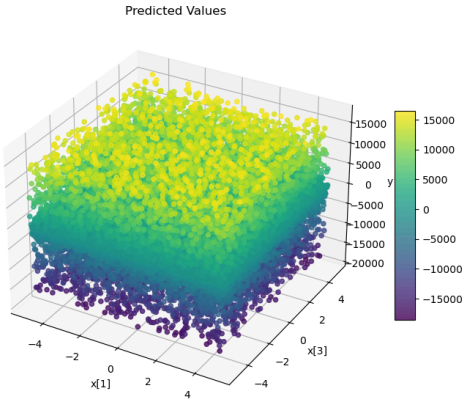

True Values
x[0]
x[5]
y

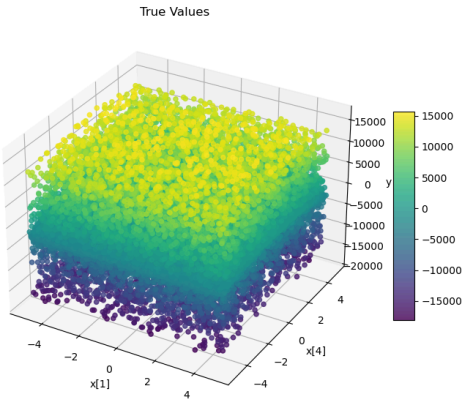

Predicted Values
x[0]
x[5]
y

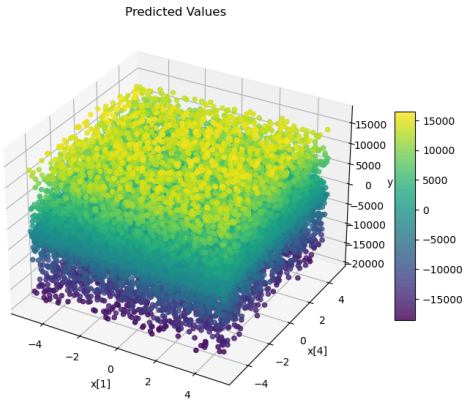

True Values
x[1]
x[2]
y

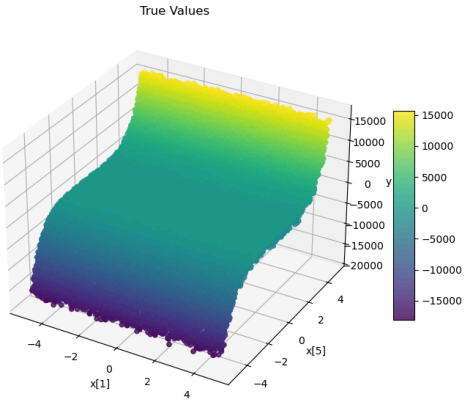

Predicted Values
x[1]
x[2]
y

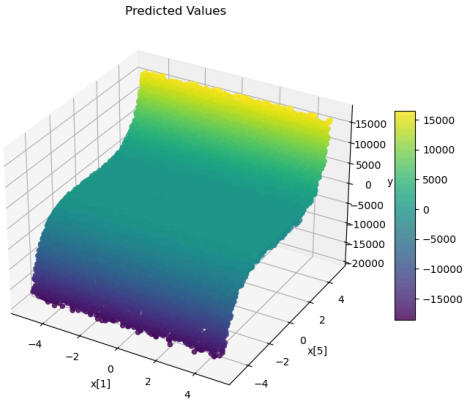

True Values
x[1]
x[3]
y

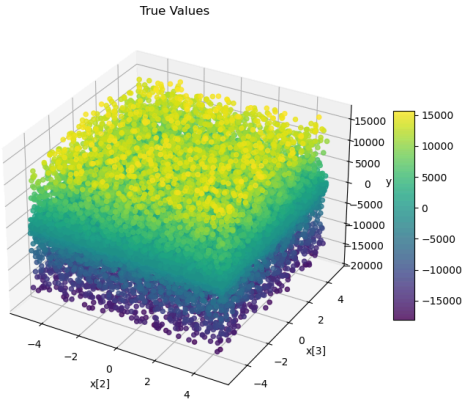

Predicted Values
x[1]
x[3]
y

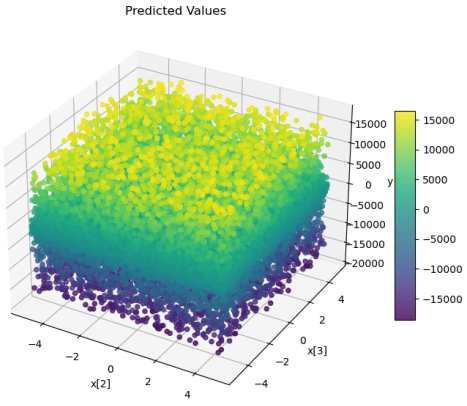

True Values
x[1]
x[4]
y

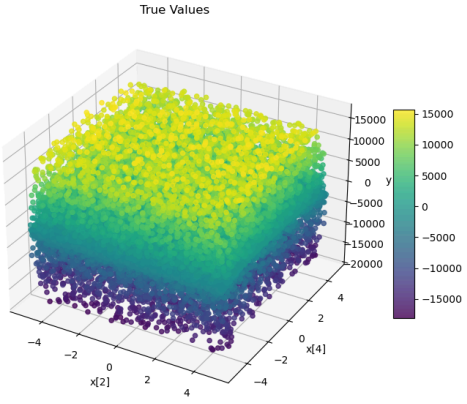

Predicted Values
x[1]
x[4]
y

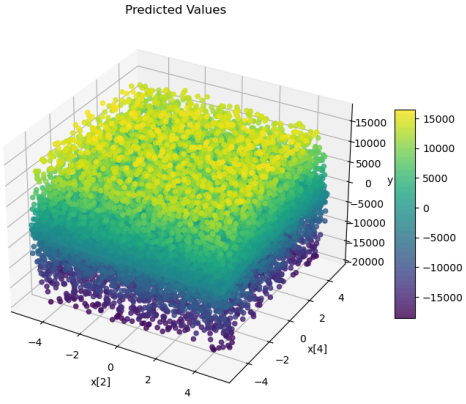

True Values
x[1]
x[5]
y

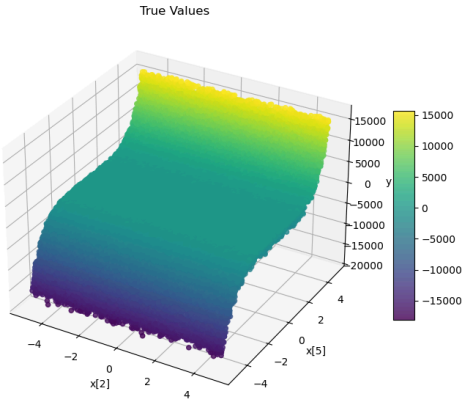

Predicted Values
x[1]
x[5]
y

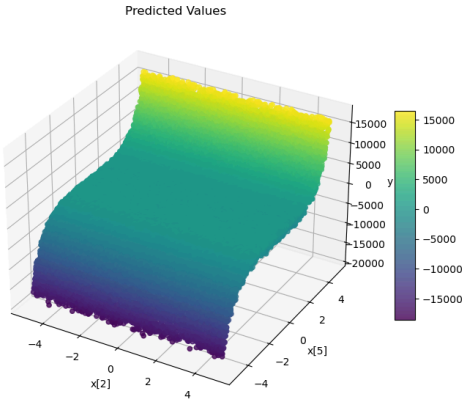

True Values
x[2]
x[3]
y

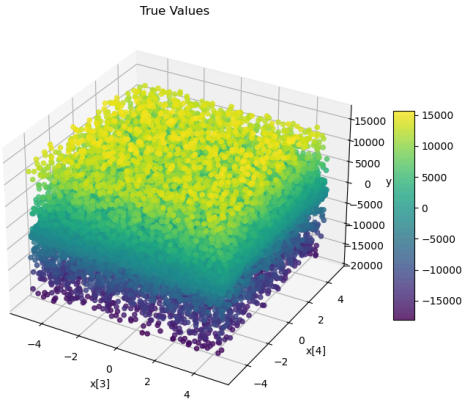

Predicted Values
x[2]
x[3]
y

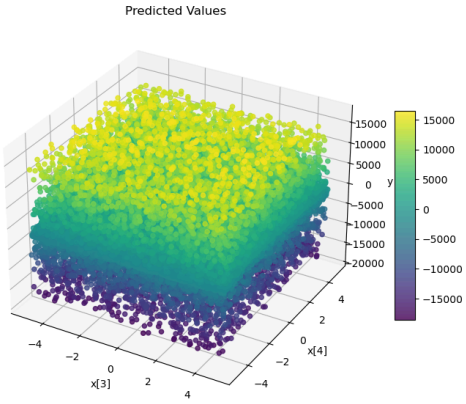

True Values
x[2]
x[4]
y

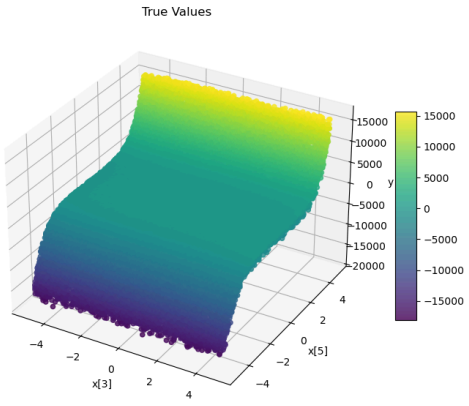

Predicted Values
x[2]
x[4]
y

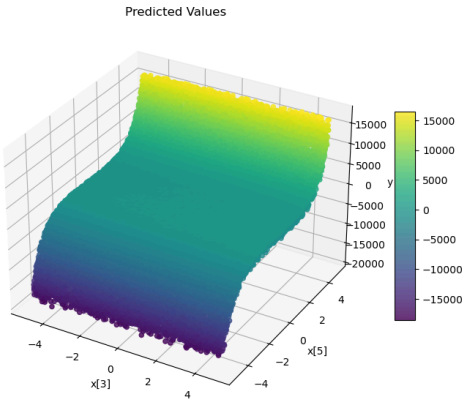

True Values
x[2]
x[5]
y

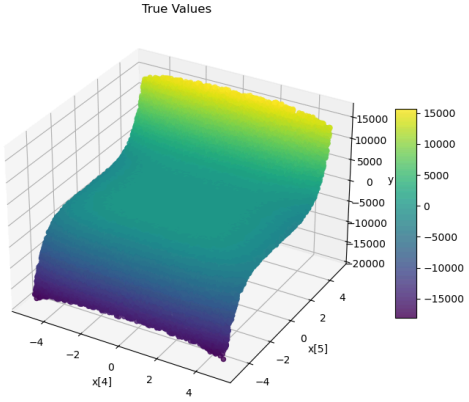

Predicted Values
x[2]
x[5]
y

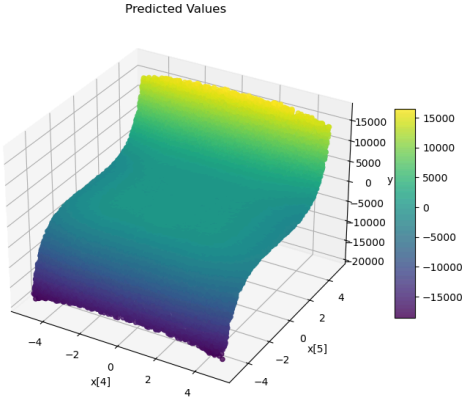

True Values
x[3]
x[4]
y


Predicted Values
x[3]
x[4]
y


True Values
x[3]
x[5]
y


Predicted Values
x[3]
x[5]
y


True Values
x[4]
x[5]
y


Predicted Values
x[4]
x[5]
y