|  |  |
| --- | --- |
| **Assignment** | |
| **Course Code** | CSC401A |
| **Course Name** | Computational Intelligence |
| **Programme** | B.Tech |
| **Department** | CSE |
| **Faculty** | FET |

|  |  |
| --- | --- |
| **Name of the Student** | Satyajit Ghana |
| **Reg. No.** | 17ETCS002159 |
| **Semester/Year** | 07/2020 |
| **Course Leader(s)** | Mr. Sagar U. |



|  |  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- | --- |
| Declaration Sheet | | | | | | | | |
| Student Name | Satyajit Ghana | | | | | | | |
| Reg. No | 17ETCS002159 | | | | | | | |
| Programme | B.Tech | | | | | Semester/Year | 07/2020 | |
| Course Code | CSC401A | | | | | | | |
| Course Title | Computational Intelligence | | | | | | | |
| Course Date |  | | to |  | | | | |
| Course Leader | Mr. Sagar U. | | | | | | | |
| **Declaration**  The assignment submitted herewith is a result of my own investigations and that I have conformed to the guidelines against plagiarism as laid out in the Student Handbook. All sections of the text and results, which have been obtained from other sources, are fully referenced. I understand that cheating and plagiarism constitute a breach of University regulations and will be dealt with accordingly. | | | | | | | | |
| Signature of the Student | |  | | | | | Date |  |
| Submission date stamp  (by Examination & Assessment Section) | |  | | | | | | |
| Signature of the Course Leader and date | | | | | Signature of the Reviewer and date | | | |
|  | | | | |  | | | |

# Contents

[Declaration Sheet ii](#_Toc58114949)

[Contents iii](#_Toc58114950)

[List of Figures iv](#_Toc58114951)

[1 Question 1 5](#_Toc58114952)

[1.1 Pitfalls in Traditional AI 5](#_Toc58114953)

[1.2 Synergism of CI Tools 7](#_Toc58114954)

[1.2.1 Computational Models of Neural Nets 7](#_Toc58114955)

[1.2.2 Genetic Algorithms 9](#_Toc58114956)

[2 Question 2 11](#_Toc58114957)

[2.1 Key ideas in hill-climbing approach 11](#_Toc58114958)

[2.2 Key ideas in alternative CI approach 12](#_Toc58114959)

[2.3 Python program demonstration 13](#_Toc58114960)

[3 Question 3 14](#_Toc58114961)

[3.1 Discussion on genetic algorithm and benchmark functions 14](#_Toc58114962)

[3.2 Python program showing the minimized values 15](#_Toc58114963)

[3.3 Comparison with any other heuristic algorithm 15](#_Toc58114964)

[Bibliography 16](#_Toc58114965)

# List of Figures

[Figure 1‑1 Retinotopic Mapping of a Cat's Cortex 8](#_Toc58114966)

[Figure 1‑2 An electrical equivalent of the biological neuron 8](#_Toc58114967)

[Figure 1‑3 The cycle of genetic algorithms 9](#_Toc58114968)

# Question 1

Solution to Question No. 1 Part A

## Pitfalls in Traditional AI

Traditional problem-solving methods in AI are concerned with representation of the problem states by symbols, and construction of a set of rules to describe the transitions in problem states. The states of the problem are then matched against the IF part of the IF-THEN rules, and on successful matching the selected rule is fired causing a transition of the existing state to a new state as obtained from the THEN part of the fired rule. To keep the rules firable until the goal is found, the knowledge base, however, calls for more search time and thus is responsible for degradation in efficiency of a reasoning system. One approach to circumvent this problem is to organize the knowledge base with fewer rules but to allow partial matching of the problem states with the IF-part of the rules. The *logic of fuzzy sets* is capable of such partial matching.

Traditional AI is very good in inductive and analogy-based learning but it is inefficient to realize supervised learning. In supervised learning, the trainer provides a number of training input/output instances for the learning system. The learning system has to adapt its internal parameters so as to generate the correct output instance in response to a given input instance. Neural Networks can do this task really well.

Except the heuristic search algorithm, traditional AI is not much competent to handle real world optimization problems. But the need for optimization of system parameters and resources in design, synthesis, diagnosis and scheduling problems, for example, is gradually increasing. Genetic Algorithm, fortunately is a new tool that has a good potential in optimizing the parameters of intelligent systems.

Decision making in real-time system largely depends on the available facts, their level of precision, the knowledge base and its soundness. A sound knowledge base should not generate false inferences, but the degree of precision of the inferences may degrade when the input facts are less precise. However, when the facts are available from multiple sources, their level of precision can be improved through the process of data fusion. Traditional AI is not concerned with data fusion. Fortunately, there is a vast contemporary literature on data fusion technology. The classical *Bayesian statistics, Dempster-Shafer theory, Pearl’s belief networks, Kalman filtering,* and *neural network* based methods are some of the well-known techniques of data fusion. These tools and techniques have successfully been used in many industrial autonomous systems. For example, noisy sensory data received by various transducers of a mobile robot are fused to take decisions about is direction of motion in a constrained obstacle-map. The other application of data fusion technique include object recognition from multi-sensory noisy data, prediction of earthquake, storm or heavy rain from multi-sensory measurements and automated diagnosis of a complex system from noisy measurements.

Traditional AI was incompetent to serve the increasing demand of search, optimization and machine learning in i) information systems with large biological and commercial databases and ii) factory automation for steel, aerospace, power and pharmaceutical industries. The failure of classical AI opened up new avenues for the non-conventional models in various engineering applications. These computational tools gave rise to a new discipline called **computational intelligence**. (A. Konar, 2006)

*“A system is computationally intelligent when it: deals with only numerical (low level) data, has pattern recognition components, does not use knowledge in the AI sense; and additionally when it (begins to) exhibit i) computational adaptivity, ii) computational fault tolerance, iii) speed approaching human-like turnaround and iv) error rates that approximate human performance.” – Prof. James (Jim) Bezdek*

## Synergism of CI Tools

To understand and elaborate on the Synergism of CI Tools we’ll start by defining CI, In an attempt to define computational intelligence, Robert J. Marks clearly mentions the name of the constituent members of the family. According to him,

*“… neural networks, genetic algorithms, fuzzy systems, evolutionary programming and artificial life are the building blocks of computational intelligence.”*

Out of these 5 members the last 2 emerged very recently, roughly speaking, and evolutionary programing is a new programming paradigm where a number of individual computer programs evolve using the concepts of genetic algorithm.

For the sake of brevity, we’ll try to understand neural networks and genetic algorithms as our CI tools.

### Computational Models of Neural Nets

Neurons are the fundamental building blocks of a biological nervous system. Typical estimates of the number of neurons in the human brain are in the order of 10 to 500 billion. Neurons are arranged into about 1,000 main modules, each with about 500 neural networks. Each network has on the order of 100,000 neurons. The axon of each neuron connects to anywhere from hundreds to thousands of other neurons; the value varies greatly from neuron to neuron and from neuron type to neuron type.

While the processing element in an artificial neural network (ANN) is generally considered to be the very roughly analogous to a biological neuron, the cell body in an ANN is modeled by a linear activation function. The activation function in general, attempts to enhance the signal contribution received through different dendrons.

But how do we know that our human neuron system uses a firing function to start the neural process? There was this Retinotopic Mapping performed on a Cat, basically a cat was strapped to a place, and the brain was injected with a radioactive fluid, now an image was shown to the cat, and it was killed as soon as the cat saw the image, the cat’s brain was laid out flat and X-Ray was taken of it, to a surprise the image had a complete print over the brain, as shown below,

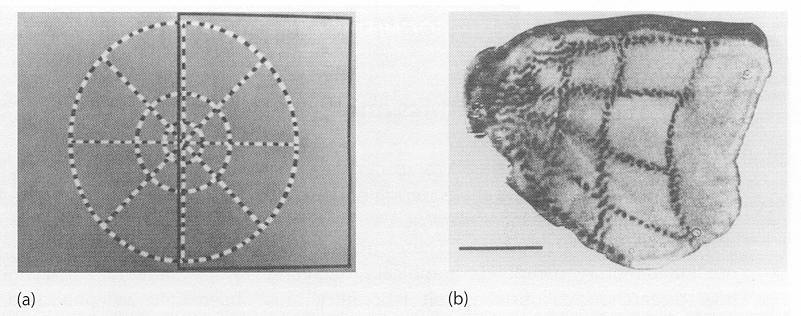


Figure ‑ Retinotopic Mapping of a Cat's Cortex

This experiment paved the way that Neural Networks would be the future of Artificial Intelligence, even in our modern neural network architectures for computer vision specifically, the entire input image is fed to the network, and several convolutions are performed to make sense of the input and get some meaningful output.

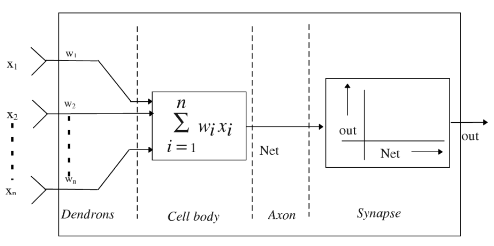


Figure ‑ An electrical equivalent of the biological neuron

ANNs can learn facts (representation by patterns) and determine the inter-relationship among the patterns. The above diagram is a simple representation of a ANN Neuron, which is very much equivalent to that of a human neuron.

### Genetic Algorithms

Genetic algorithm (GA) is a stochastic algorithm that models the evolutionary process of biological species through natural selection. Proposed by Holland in early 60’s, this algorithm is gaining its importance for its wide acceptance in solving three classical problems, such as learning, searching and optimizing.

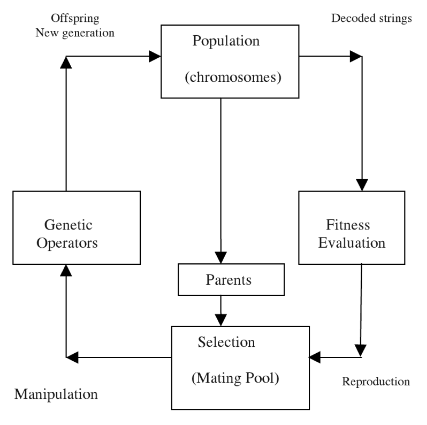


Figure ‑ The cycle of genetic algorithms

A GA operated through a simple cycle of stages:

1. Creation of a “population” of strings
2. Evaluation of each string
3. Selection of best strings and
4. Genetic manipulation to create new population of strings

Each cycle in GA produces a new generation of possible solutions for a given problem. In the first phase, an initial population, describing representatives of the potential solution, is created to initialize the search process. The elements of the population are encoded into bit strings, called chromosomes. The performance of the strings, often called fitness, is then evaluated. Depending upon the fitness of the chromosomes, they are selected for a genetic manipulation process. In the first step, crossover operation combines two individuals about a crossover point, basically their chromosomes are swapped and two new individuals are created. The crossover point is chosen at random. The next step is mutation, in which random places in the offsprings’s chromosome are mutated or flipped.

# Question 2

Solution to Question No. 1 Part B

## Key ideas in hill-climbing approach

The hill-climbing search algorithm (steepest-ascent version), is simply a loop that continually moves in the direction of increasing value—that is, uphill. It terminates when it reaches a “peak” where no neighbor has a higher value. The algorithm does not maintain a search tree, so the data structure for the current node need only record the state and the value of the objective function. Hill climbing does not look ahead beyond the immediate neighbors of the current state. This resembles trying to find the top of Mount Everest in a thick fog while suffering from amnesia

Hill climbing is sometimes called *greedy local search* because it grabs a good neighbor state without thinking ahead about where to go next. Although greed is considered one of the seven deadly sins, it turns out that greedy algorithms often perform quite well. Hill climbing often makes rapid progress toward a solution because it is usually quite easy to improve a bad state.

**function** HILL-CLIMBING(problem) **return**s a state that is a local minimum

current ← MAKE-NODE(problem.INITIAL-STATE)

**loop do**

neighbour ← a highest-values successor of current

if neighbour.VALUE <= current.VALUE **then return** current.STATE

current ← neighbour

Some key things that Stochastic Hill Climbing does, but explained like a layman

* close your eyes, and just move, if you moved and can feel that you are at a higher step, then stay there
* if you are stuck in a local-minima you are stuck there forever, period.
* you can try to take longer steps to get out of that minima, but that will make you move randomly and you will end up brute-forcing the entire search space
* if the search space has a single global-minima, you’ll be able to reach there pretty quickly, imagine like running down a mountain, its easy right?

## Key ideas in alternative CI approach

A quick and dirty fix to the pitfalls of hill climbing is Simulated Annealing, A hill-climbing algorithm that never makes “downhill” moves toward states with lower value (or higher cost) is guaranteed to be incomplete, because it can get stuck on a local maximum. In contrast, a purely random walk—that is, moving to a successor chosen uniformly at random from the set of successors—is complete but extremely inefficient. Therefore, it seems reasonable to try to combine hill climbing with a random walk in some way that yields both efficiency and completeness. Simulated annealing is such an algorithm.

In metallurgy, annealing is the process used to temper or harden metals and glass by heating them to a high temperature and then gradually cooling them, thus allowing the material to reach a low-energy crystalline state. To explain simulated annealing, we switch our point of view from hill climbing to gradient descent (i.e., minimizing cost) and imagine the task of getting a ping-pong ball into the deepest crevice in a bumpy surface. If we just let the ball roll, it will come to rest at a local minimum. If we shake the surface, we can bounce the ball out of the local minimum. The trick is to shake just hard enough to bounce the ball out of local minima but not hard enough to dislodge it from the global minimum. The simulated-annealing solution is to start by shaking hard (i.e., at a high temperature) and then gradually reduce the intensity of the shaking (i.e., lower the temperature).

**function** SIMULATED-ANNEALING(problem, schedule) returns a solution state

**inputs**: problem, a problem

schedule, a mapping from time to “temperature”

current ← MAKE-NODE(problem.INITIAL-STATE)

**for** t = 1 **to** ∞ do

T ← schedule(t)

**if** T = 0 **then return** current

next ← a randomly selected successor of current

ΔE ← next.VALUE – current.VALUE

**if** ΔE > 0 **then** current ← next

**else** current ← next only with probability

## Python program demonstration

In order to test the algorithm’s optimization capability, they were subjected to two types of functions, one having a single global minima, and one with many local minima but with a single global minima.

Why? Because real world problems don’t necessarily are smooth and have a single minima, instead its highly nonlinear with many local minima.

What about maxima finding? Well, all maxima finding optimizations can be converted to minima finding by simply negating the value.

Sphere Function 1:

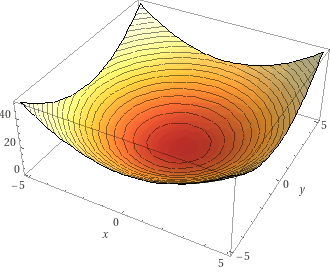


Figure ‑ environment function x^2+y^2

Sphere Function 2:

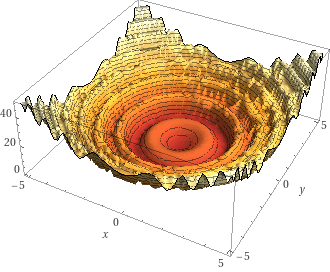


Figure ‑ environment function 5\*sin(x^2+y^2)+x^2+y^2

Source code for Hill Climbing Algorithm:

NOTE: francium is a self-made library for testing ai algorithms given an environment and an agent, the details to which can be found in Appendix.

**agent.py**

*from* francium.algorithms.hill\_climbing.environment *import* Environment

*from* francium.core *import* BaseAgent, State, BaseEnvironment

*import* numpy *as* np

*class* Agent(BaseAgent):

    """Hill Climbing AI Agent"""

*def* \_\_init\_\_(self, step\_size: float = 1e-2):

*self*.step\_size = step\_size

*def* act(self, state: State, env: Environment):

*# get the current state, act upon it and give a new state*

        new\_state = State({"x": state["x"], "y": state["y"]})

        new\_state["x"] += np.random.randn() \* *self*.step\_size

        new\_state["y"] += np.random.randn() \* *self*.step\_size

*# check if you are in bounds of the environment*

        new\_state["x"] = min(max(new\_state["x"], env.x\_bounds[0]), env.x\_bounds[1])

        new\_state["y"] = min(max(new\_state["y"], env.y\_bounds[0]), env.y\_bounds[1])

*return* new\_state

**environment.py**

*from* typing *import* Tuple, Optional

*from* francium.core *import* BaseEnvironment, State

*from* francium.core.eval\_functions *import* \*

*class* Environment(BaseEnvironment):

*def* \_\_init\_\_(

        self,

        x\_bounds: Tuple[float, float],

        y\_bounds: Tuple[float, float],

        goal\_val: Optional[State] = *None*,

        tolerance: Optional[float] = 1e-4,

        eval\_func=convex\_x\_square,

    ):

        BaseEnvironment.\_\_init\_\_(

*self*, x\_bounds, y\_bounds, goal\_val, tolerance, eval\_func=eval\_func

        )

*def* evaluate\_state(self, state: State) -> Tuple[float, bool]:

        eval\_val = *self*.evaluation\_func(state["x"], state["y"])

*if* *self*.goal\_val:

            is\_done = (

*True*

*if* np.abs(eval\_val - *self*.goal\_val["z"]) < *self*.tolerance

*else* *False*

            )

*else*:

            is\_done = *False*

*return* eval\_val, is\_done

**solver.py**

*from* typing *import* Optional

*from* francium.algorithms.hill\_climbing.agent *import* Agent

*from* francium.algorithms.hill\_climbing.environment *import* Environment

*from* francium.core *import* BaseSolver, State, setup\_logger

logger = setup\_logger(\_\_name\_\_)

*class* Solver(BaseSolver):

*def* \_\_init\_\_(self, agent: Agent, environment: Environment):

        BaseSolver.\_\_init\_\_(*self*, agent, environment)

*self*.initialized: bool = *False*

*def* init\_solver(self, init\_state: Optional[State] = *None*):

*if* init\_state:

*self*.memory.add\_episode(init\_state)

*else*:

            init\_state = *self*.env.get\_random\_init\_position()

        logger.info(f"=> Initialized Solver with State: {init\_state}")

*self*.memory.add\_episode(init\_state)

*self*.initialized = *True*

*def* train\_step(self) -> bool:

*if* not *self*.initialized:

            logger.error("=> Solver not initialized !")

*raise* Exception("Initialize the solver `solver.init\_solver()`")

        curr\_state = *self*.memory.get\_curr\_state()

        new\_state = *self*.agent(curr\_state, *self*.env)

        eval\_val, is\_done = *self*.env.evaluate\_state(new\_state)

*if* is\_done:

            logger.info(

                f"=> training is done ! best state: {*self*.memory.get\_curr\_state()}"

            )

*return* *False*

*if* eval\_val < curr\_state["z"]:

*# logger.info(f"z: f(x = {new\_state['x']}, y = {new\_state['y']}) = {eval\_val}")*

            new\_state["z"] = eval\_val

*self*.memory.add\_episode(new\_state)

*return* *True*

Running the Hill Climbing Algorithm

import francium.algorithms.hill\_climbing as hc

import francium.core.eval\_functions as eval\_functions

from francium.core import State

agent = hc.Agent(step\_size=1e-1)

env = hc.Environment(x\_bounds=(-5.0, 5.0), y\_bounds=(-5.0, 5.0), eval\_func=eval\_functions.convex\_x\_square)

solver = hc.Solver(agent=agent, environment=env)

solver.init\_solver(

    init\_state=State({

        'x': 4.0,

        'y': 2.0,

        'z': env.evaluation\_func(4.0, 2.0)

    })

)

for episode in range(1000):

    trainable = solver.train\_step()

    if not trainable:

        break

solver.memory.best\_episode

>>> {'x': -0.0003257200454512485, 'y': 0.006713925485853664, 'z': 4.518288897760412e-05}

solver.plot\_history()

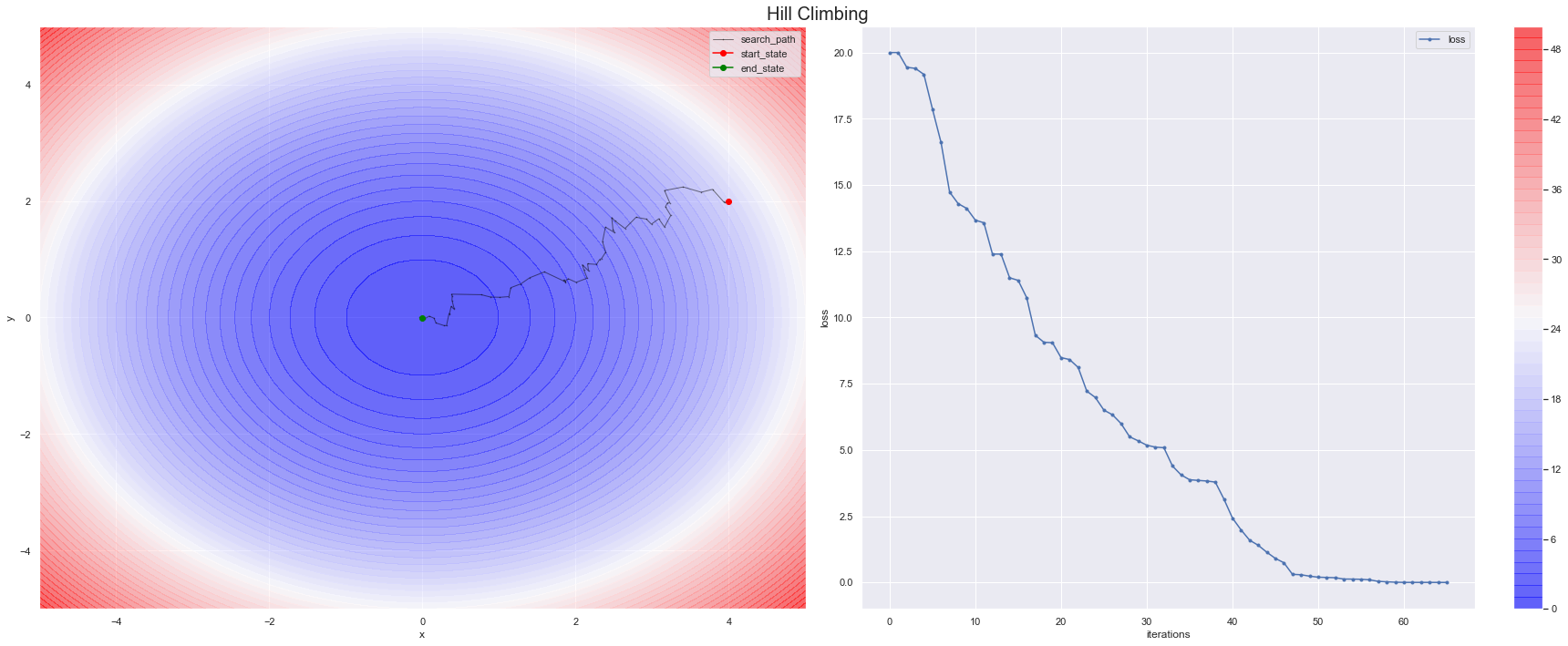
>>>

Figure ‑ hill-climbing solver on first environment

env.plot\_environment()

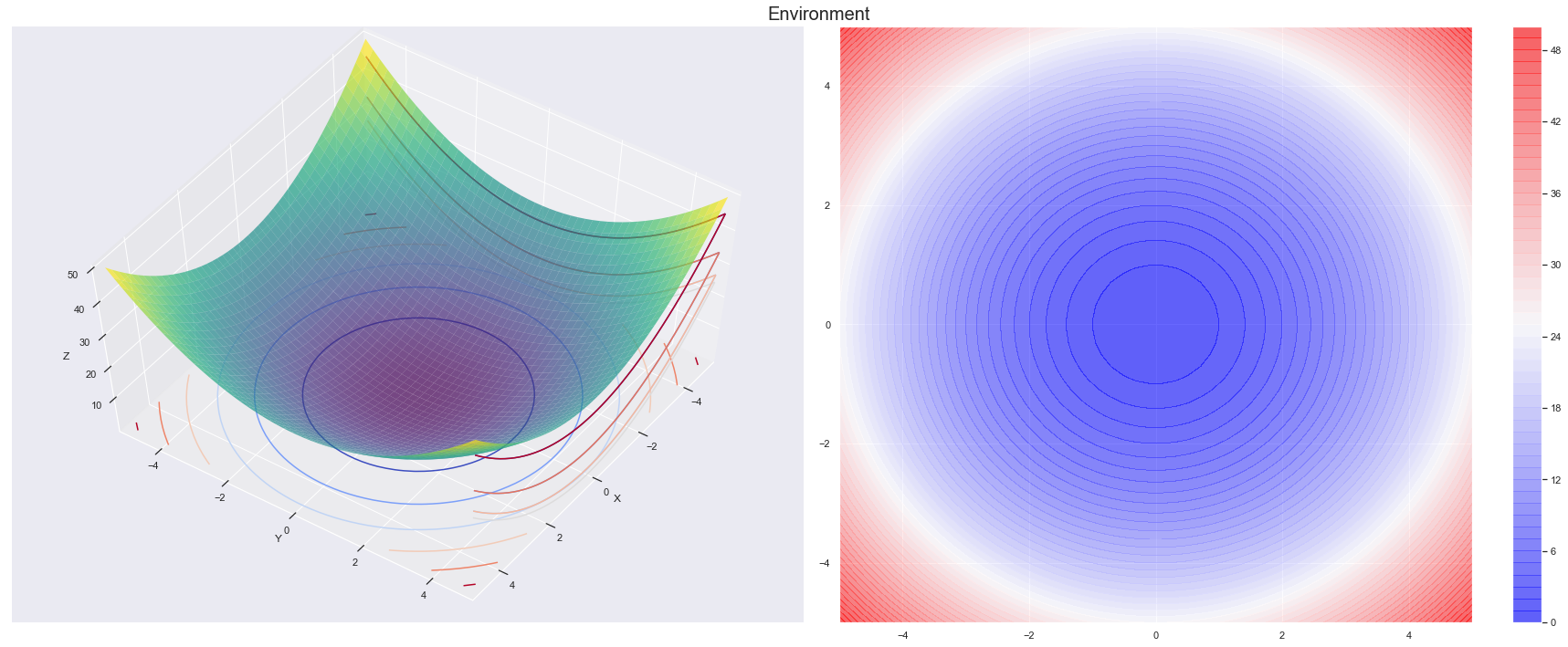
>>>

Figure ‑ hill-climbing first environment

agent = hc.Agent(step\_size=1e-1)

env = hc.Environment(x\_bounds=(-5.0, 5.0), y\_bounds=(-5.0, 5.0), eval\_func=eval\_functions.sinx\_plus\_x)

solver = hc.Solver(agent=agent, environment=env)

solver.init\_solver(

    init\_state=State({

        'x': 4.0,

        'y': 2.0,

        'z': env.evaluation\_func(4.0, 2.0)

    })

)

for episode in range(1000):

    trainable = solver.train\_step()

    if not trainable:

        break

solver.memory.best\_episode

>>> {'x': 3.4188550057829175, 'y': 2.321218803102678, 'z': 12.178423661021002}

solver.plot\_history()

>>>

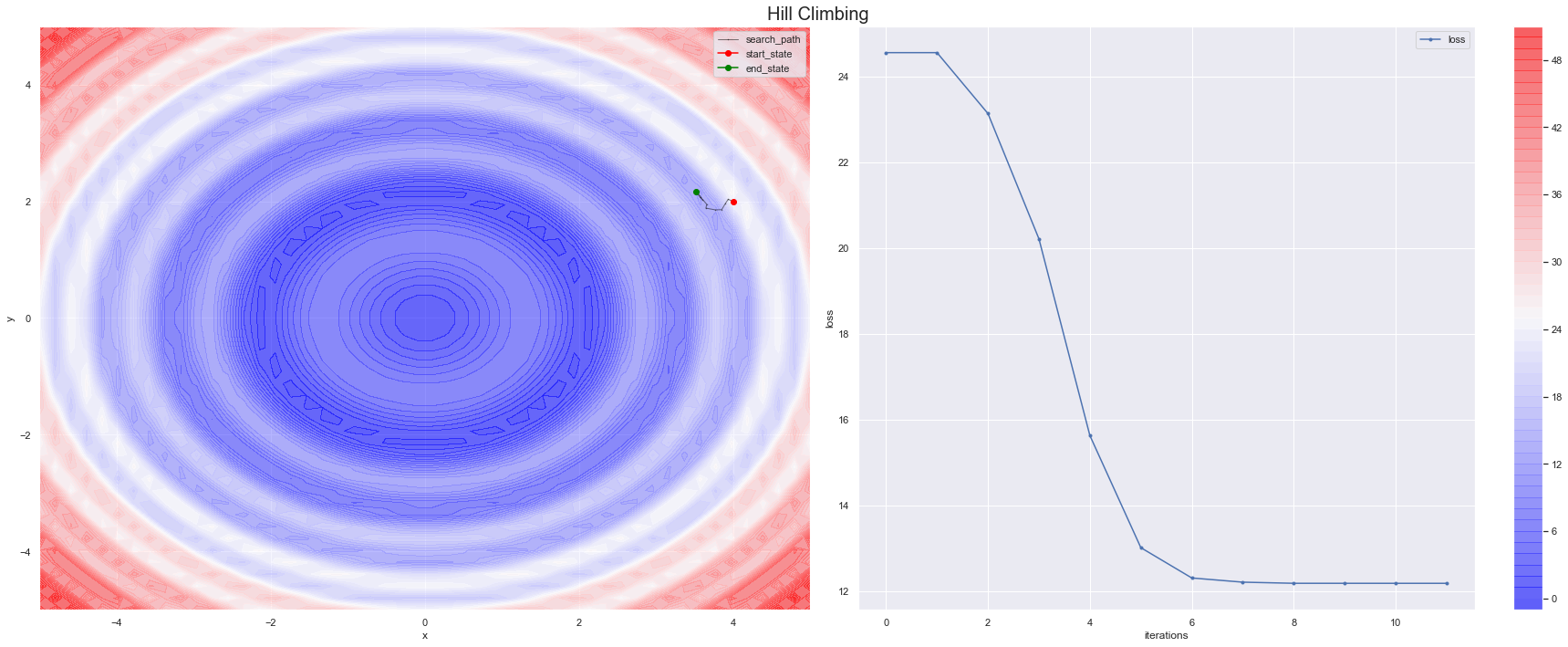
env.plot\_environment()

Figure ‑ hill-climbing solver in second environment

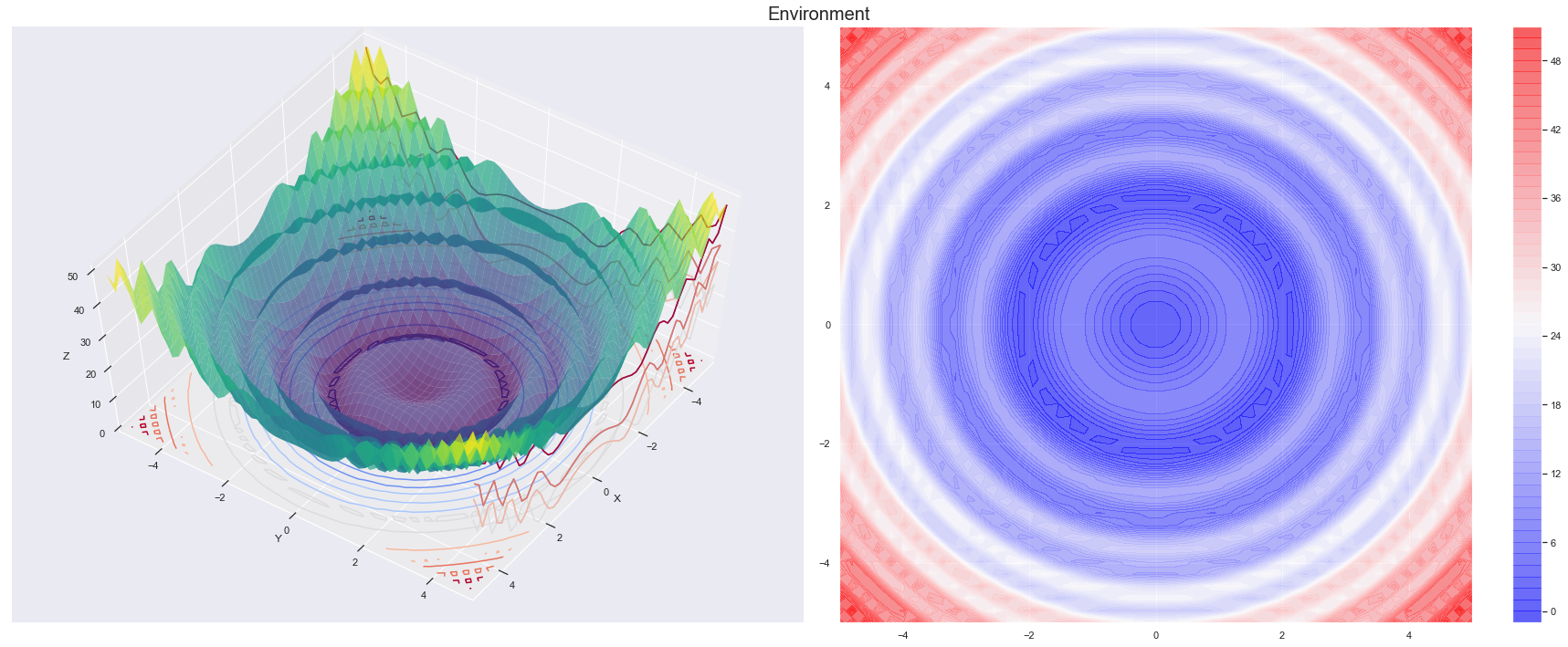
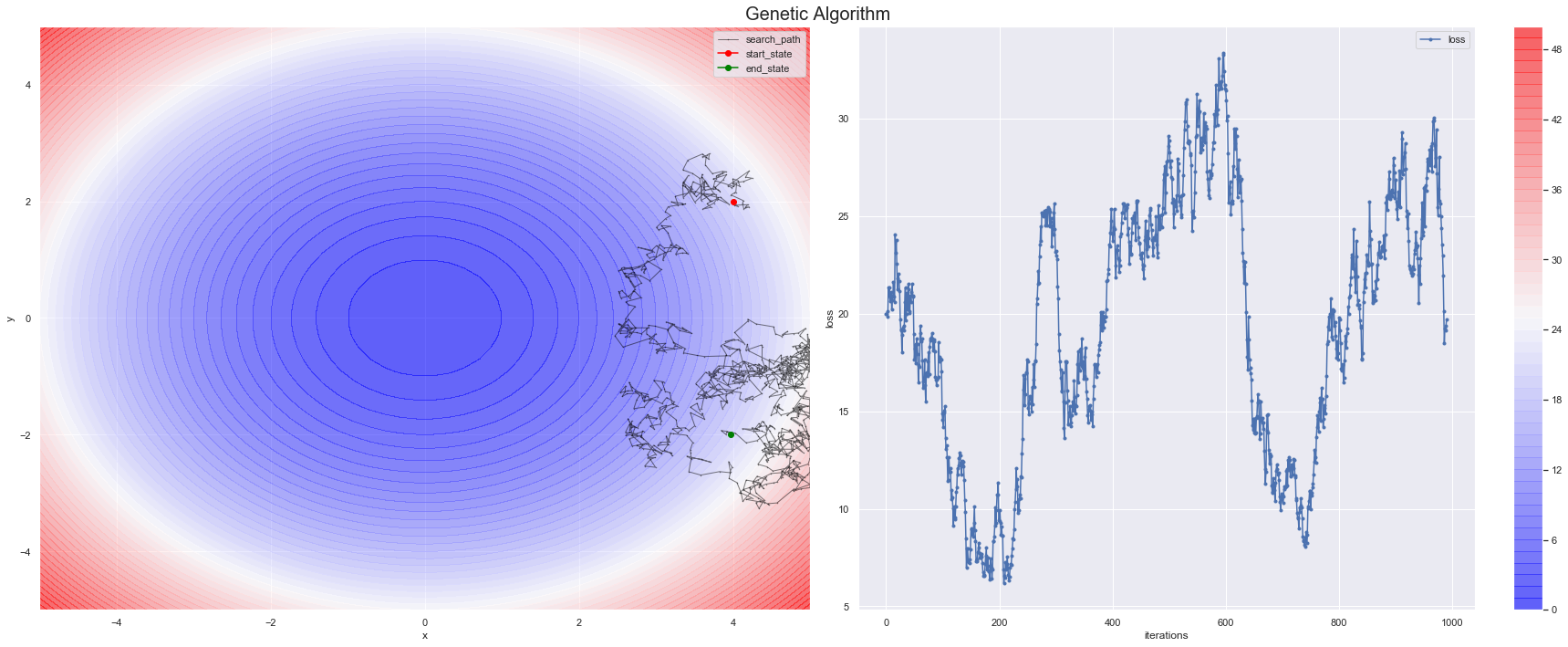
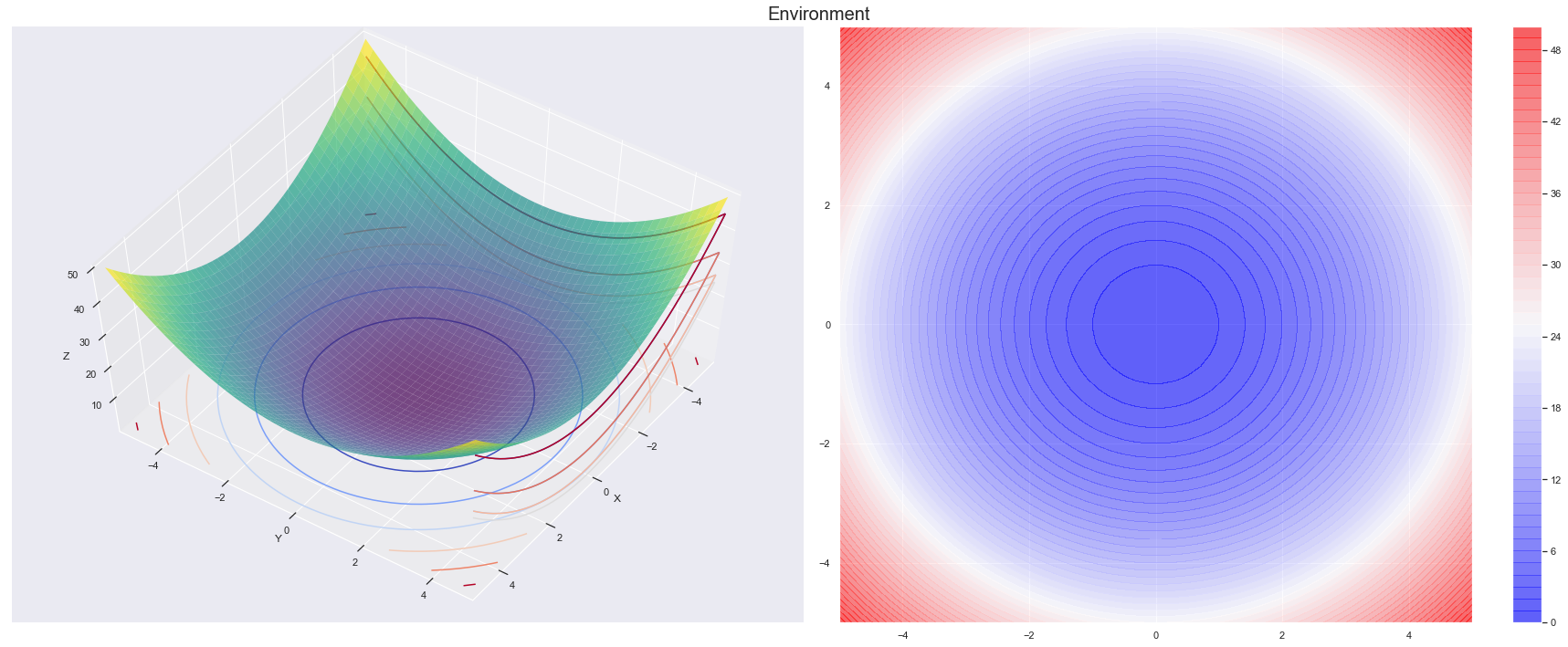
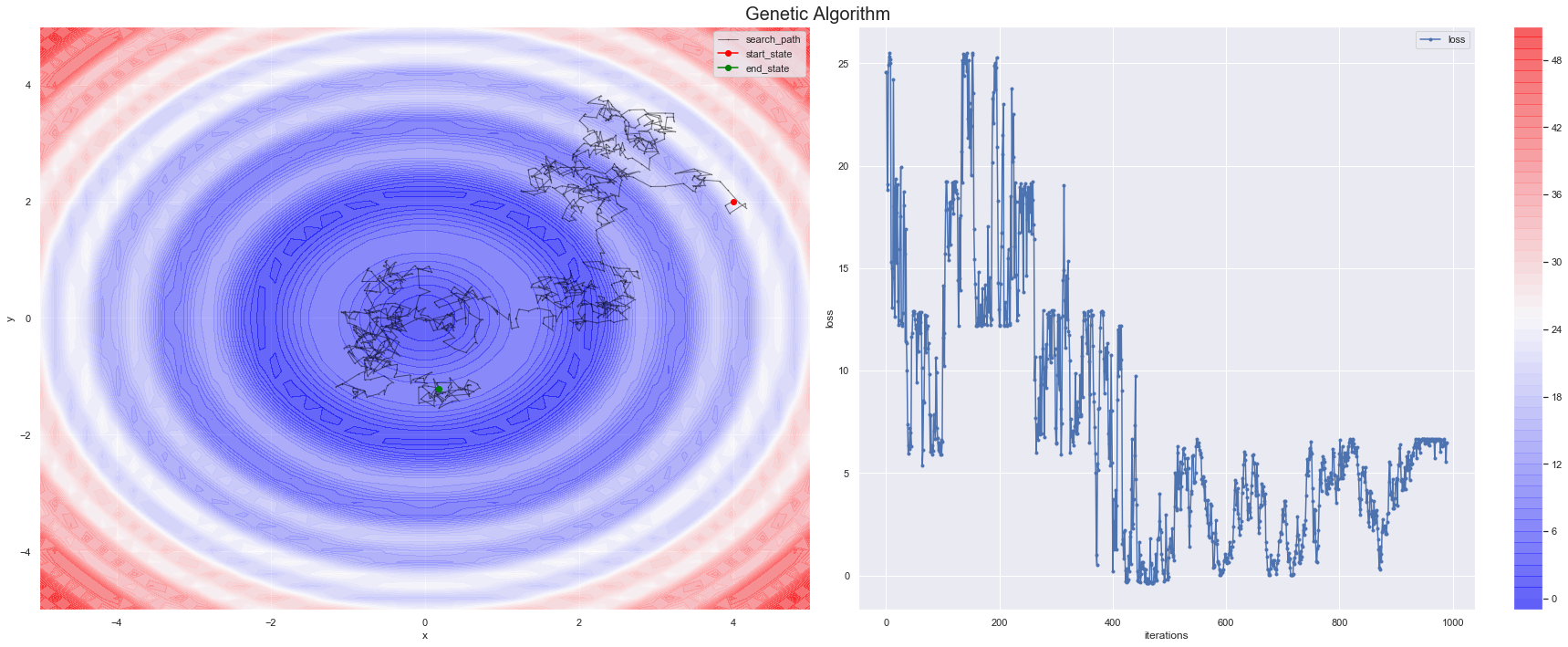
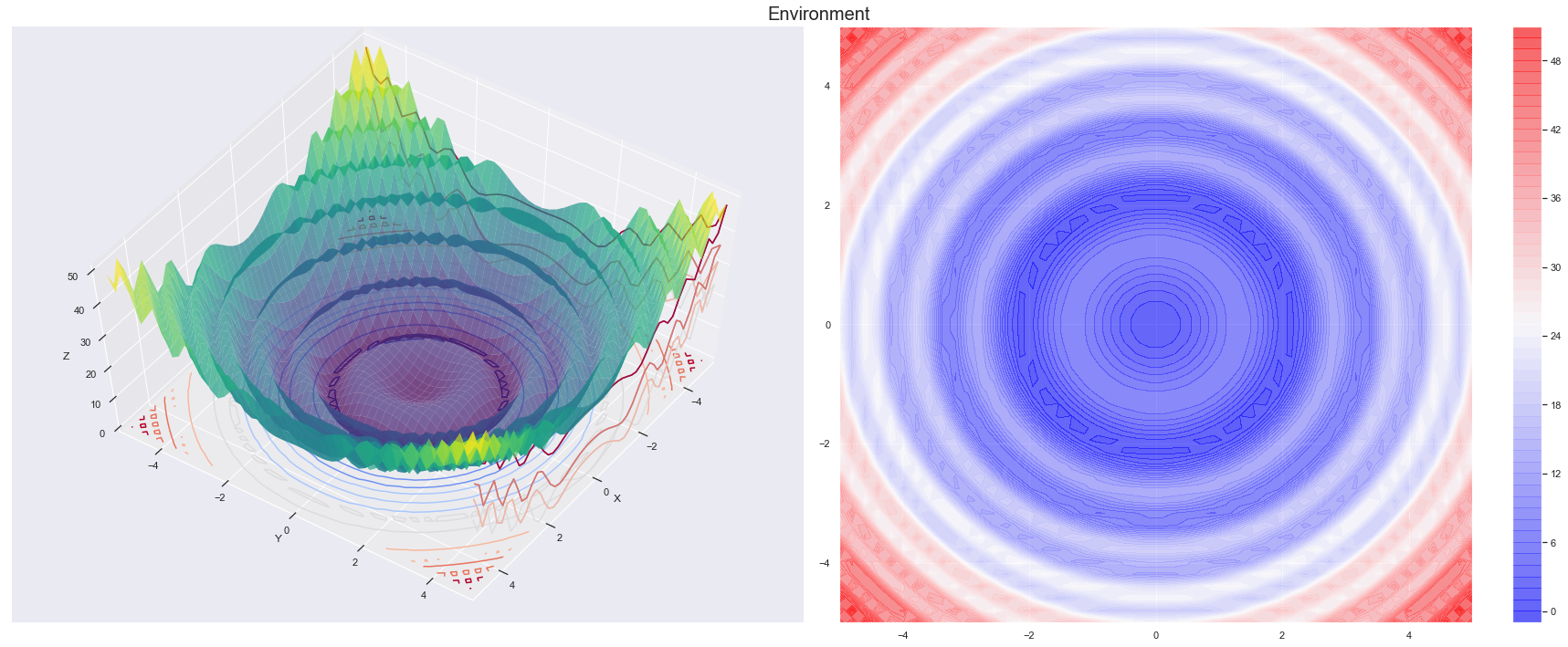
>>>

Figure ‑ hill-climbing second environment









# Question 3

Solution to Question No. 2 Part B

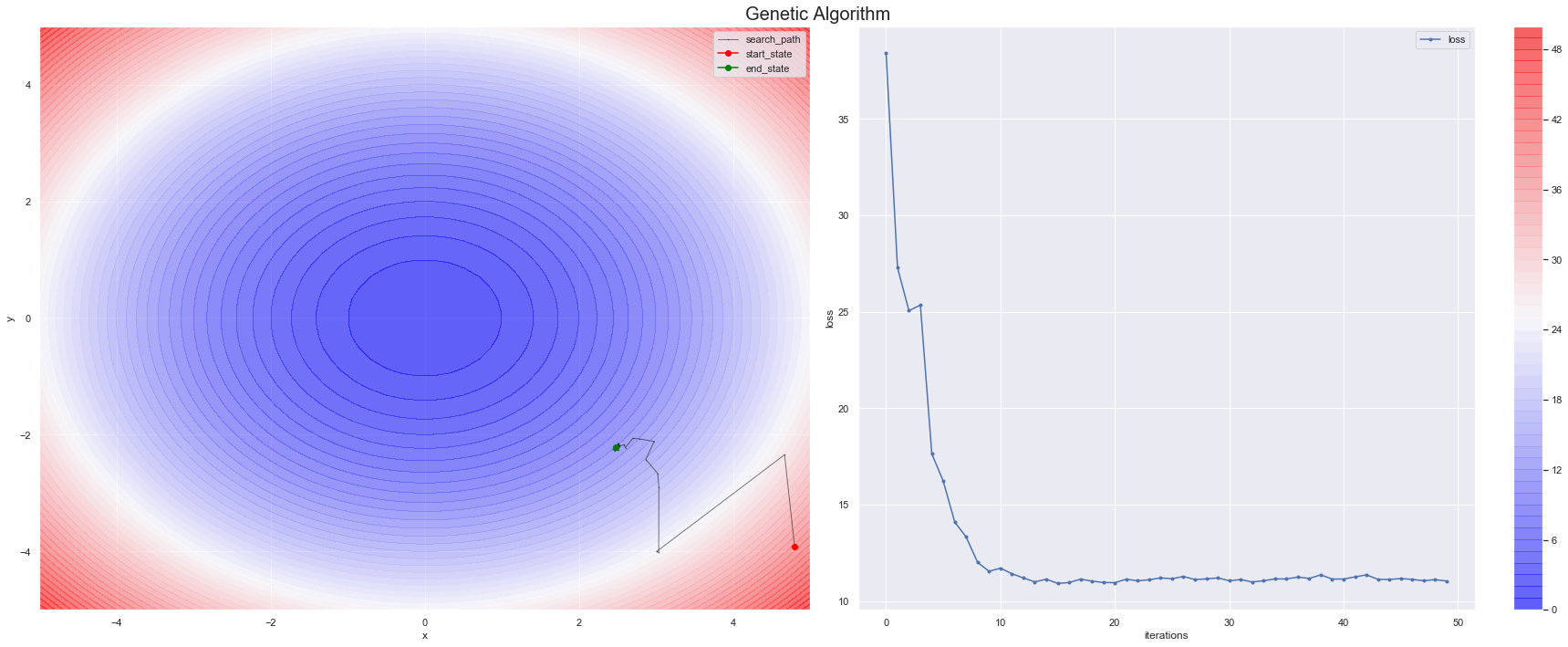
## Discussion on genetic algorithm and benchmark functions

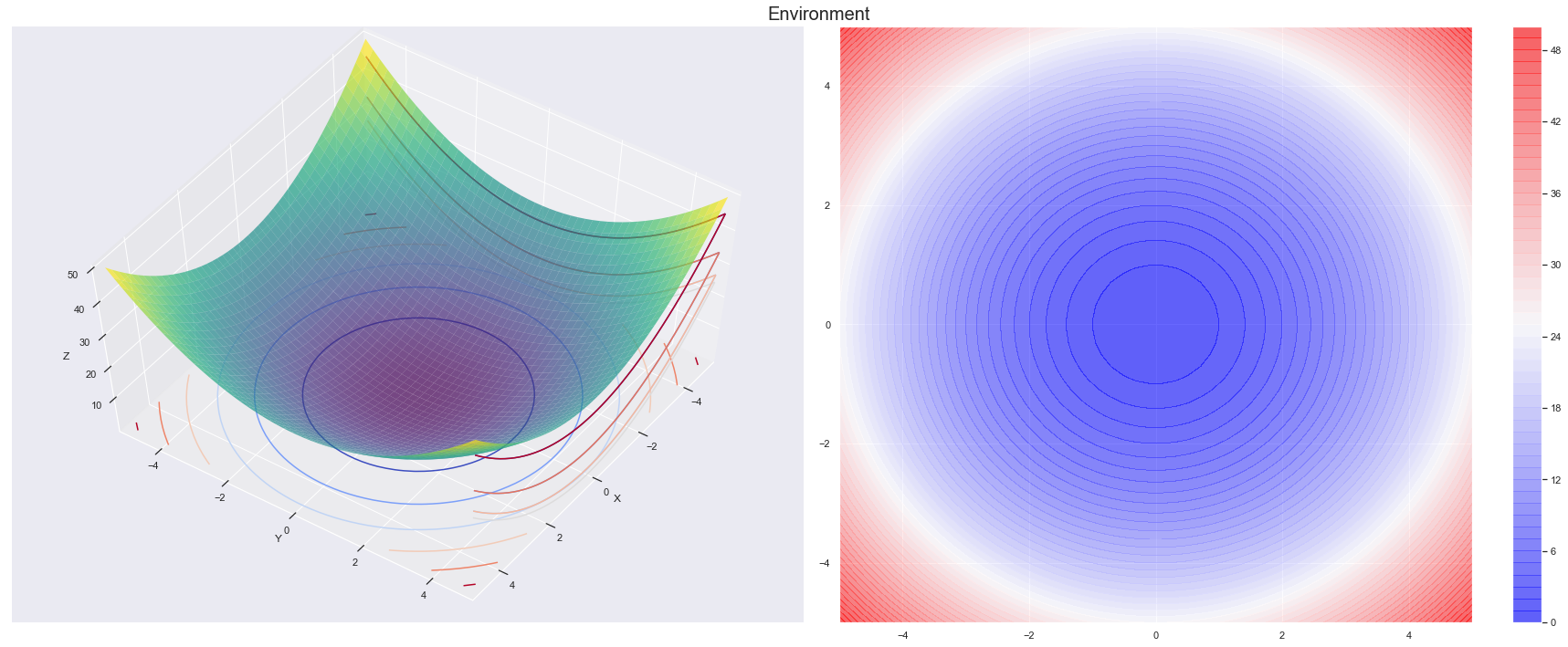
A genetic algorithm (or GA) is a variant of stochastic beam search in which successor states are generated by combining two parent states rather than by modifying a single state. The analogy to natural selection is the same as in stochastic beam search, except that now we are dealing with sexual rather than asexual reproduction.

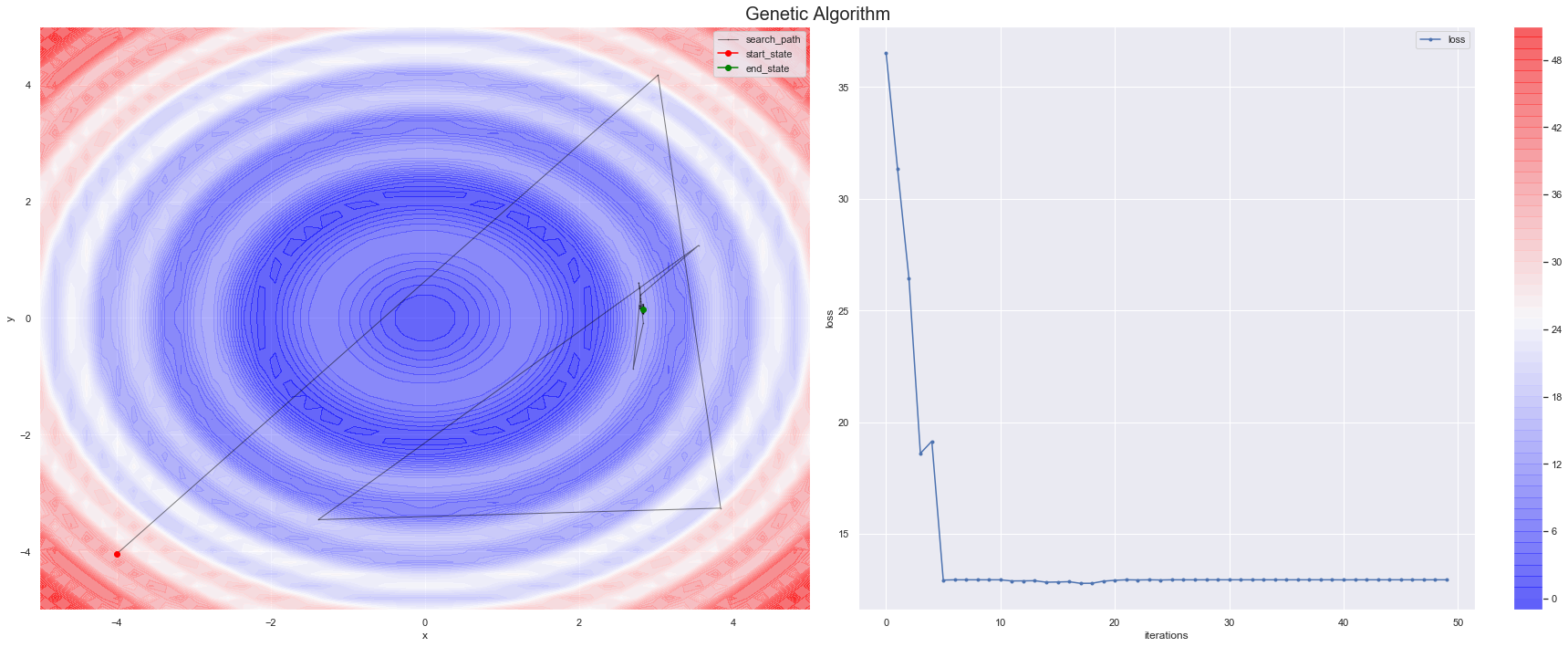
Like beam searches, GAs begin with a set of k randomly generated states, called the population. Each state, or individual, is represented as a string over a finite alphabet—most commonly, a string of 0s and 1s. Each state is rated by the objective function, or (in GA terminology) the fitness function. A fitness function should return higher values for better states. Two pairs are selected at random for reproduction, in accordance with the probabilities. For each pair to be mated, a crossover point is chosen randomly from the positions in the string. The crossover points are after the third digit in the first pair and after the fifth digit in the second pair. Finally, each location is subject to random mutation with a small independent probability.

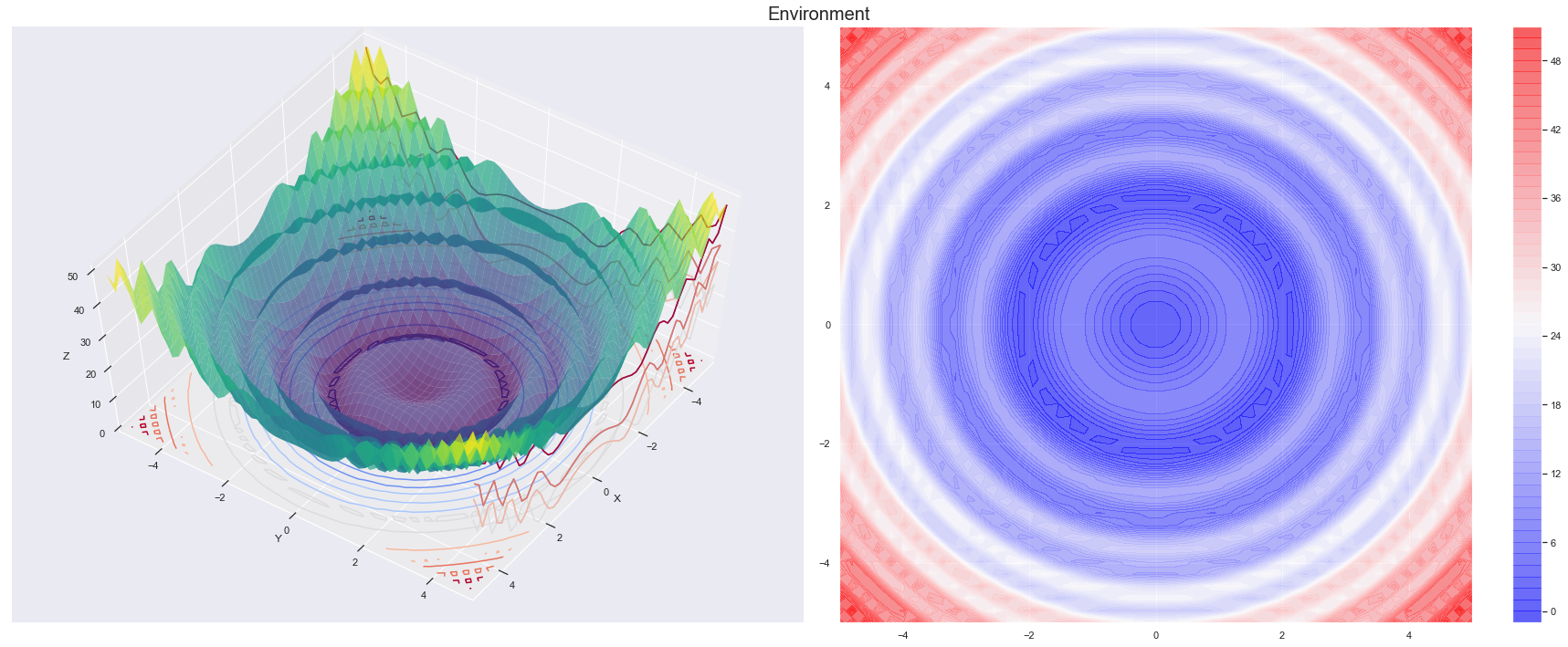
Like stochastic beam search, genetic algorithms combine an uphill tendency with random exploration and exchange of information among parallel search threads. The primary advantage, if any, of genetic algorithms comes from the crossover operation. Yet it can be shown mathematically that, if the positions of the genetic code are permuted initially in a random order, crossover conveys no advantage. Intuitively, the advantage comes from the ability of crossover to combine large blocks of letters that have evolved independently to perform useful functions, thus raising the level of granularity at which the search operates.

## Python program showing the minimized values









## Comparison with any other heuristic algorithm

# Bibliography

1. Konar, A., 2006. Computational intelligence: principles, techniques and applications. Springer Science & Business Media.
2. Klir, G.J. and Yuan, B., 1996. Fuzzy sets and fuzzy logic: theory and applications. Possibility Theory versus Probab. Theory, 32(2), pp.207-208.
3. Anderson, J.A., 1972. A simple neural network generating an interactive memory. Mathematical biosciences, 14(3-4), pp.197-220.
4. Bender, E.A., 1996. Mathematical methods in artificial intelligence.
5. Vose, M.D. and Liepins, G.E., 1991. Punctuated equilibria in genetic search. Complex systems, 5(1), pp.31-44.
6. Vose, M.D., 1999. What are genetic algorithms? A mathematical prespective. In Evolutionary algorithms (pp. 251-276). Springer, New York, NY.
7. Russell, S. and Norvig, P., 2002. Artificial intelligence: a modern approach.

# Appendix

ProjektFrancium



This assignment gave rise to ProjektFrancium, a tool for experimenting with AI Algorithms

[**https://github.com/satyajitghana/ProjektFrancium**](https://github.com/satyajitghana/ProjektFrancium)

*import* francium.algorithms.hill\_climbing *as* hc

*import* francium.core.eval\_functions *as* eval\_functions

*from* francium.core *import* State

agent = hc.Agent(step\_size=1e-1)

env = hc.Environment(x\_bounds=(-5.0, 5.0), y\_bounds=(-5.0, 5.0), eval\_func=eval\_functions.sinx\_plus\_x)

solver = hc.Solver(agent=agent, environment=env)

solver.init\_solver(

    init\_state=State({

        'x': 4.0,

        'y': 2.0,

        'z': env.evaluation\_func(4.0, 2.0)

    })

)

*for* episode *in* range(1000):

    trainable = solver.train\_step()

*if* not trainable:

*break*

solver.plot\_history()

env.plot\_environment()