ARTIFICIAL INTELLIGENCE
CS-F407

LAB PTOJECT REPORT

Instructor In Charge: - Dr. Aneesh S Chivukula

Submitted By: - SHASHANK GAUTAM (2020B5A32378H)
RAGHAV SHARMA (2020B2A42461H)
SHRESTH GATTANI (2020B1A72446H)

1ST ALGORITHM: -

1. Introduction:
In this report, we discuss and analyze the Depth-First Search (DFS) algorithm implemented to find the maximum value of an objective function. The objective function takes two variables, 'x' and 'y', as input and returns a numerical value based on a mathematical expression. The DFS algorithm explores the solution space by iteratively updating 'x' and 'y' and tracks the best solution and value found so far. The main goal is to find the combination of 'x' and 'y' that maximizes the objective function.

2. Objective Function:
The objective function is defined as follows:

```
def objective_function(x, y):
    return ((1.38 * pow(10, -23) * x) / (1.6 * pow(10, -19))) * (math.log((2.54 / 1e-11) * y))
```

The function takes two input parameters, 'x' and 'y', and calculates a value based on a mathematical expression. The expression involves basic arithmetic operations, the natural logarithm, and mathematical constants. The objective of the DFS algorithm is to find the values of 'x' and 'y' that maximize this function.

3. Depth-First Search (DFS) Algorithm:
The DFS algorithm is used to systematically explore the search space of 'x' and 'y' to find the best combination that maximizes the objective function. The algorithm operates as folows:

```
def dfs_search(max_depth, x_range, y_range):
    best_solution = None
    best_value = float('-inf')

    stack = [(0, x_range, y_range)]

    while stack:
        depth, x_range, y_range = stack.pop()

        if depth >= max_depth:
            continue

        for x in range(x_range[0], x_range[1] + 1):
            for y in range(y_range[0], y_range[1] + 1):
                value = objective_function(x, y)

                if value > best_value:
                    best_solution = (x, y)
                    best_value = value

                stack.append((depth + 1, (x, x), (y, y)))

    return best_solution, best_value
```

4. Function Explanation:

The dfs_search function takes three parameters as input:

max_depth: The maximum depth of the search tree. It limits the number of iterations the DFS algorithm will perform.
x_range: A tuple representing the range of values for 'x'.
y_range: A tuple representing the range of values for 'y'.
Inside the function, best_solution and best_value are initialized to None and negative infinity, respectively. These variables will be updated as the algorithm progresses to store the best solution found and its corresponding value.

The stack is used to simulate the DFS algorithm. It starts with a single element, representing the root node of the search tree. Each element of the stack contains the current depth, the range of 'x' values, and the range of 'y' values.

The algorithm proceeds with a while loop that continues until the stack is empty.

Inside the loop, the depth, 'x_range', and 'y_range' are extracted from the top element of the stack.

If the current depth exceeds the max_depth, the algorithm skips further exploration at this node and continues to the next element in the stack.

The algorithm iterates through all possible 'x' and 'y' values within their respective ranges.

For each combination of 'x' and 'y', the objective function is evaluated, and the value is stored in the value variable.

If the current value is greater than the current best_value, the algorithm updates the best_solution and best_value with the current 'x' and 'y' and their corresponding value.

Next, the algorithm appends new elements to the stack with increased depth and narrowed 'x' and 'y' ranges to explore the search space further.

Finally, when the search is complete, the algorithm returns the best_solution and best_value.

5. Example Usage and Output:
The code provides an example usage of the dfs_search function:

best_solution, best_value = dfs_search(max_depth=10, x_range=(283, 300), y_range=(750, 1000))

print(f"Best solution: x={best_solution[0]}, y={best_solution[1]}")
print(f"Best value: {best_value}").

In this example, the DFS algorithm searches for the best combination of 'x' and 'y' with a maximum depth of 10 and the specified ranges for 'x' (283 to 300)

and 'y' (750 to 1000). The results, i.e., the best solution and the corresponding value, are printed to the console.

6. Conclusion:

The DFS algorithm provided in the code efficiently searches the solution space of the objective function and identifies the combination of 'x' and 'y' that maximizes the function. By setting the appropriate max_depth and ranges for 'x' and 'y', the algorithm can be fine-tuned to find optimal solutions in various scenarios. However, it's important to note that the algorithm's performance heavily depends on the search space and the complexity of the objective function. For more complex problems, other search algorithms like genetic algorithms or gradient-based methods might be more suitable.

2ND ALGORITHM: -

Report on Hill Climbing Algorithm and New Hill Climbing Algorithm

1. Introduction:
In this report, we will explore and analyze two optimization algorithms, Hill Climbing and New Hill Climbing, used to maximize the output of an objective function. The objective function takes two variables, 'x' and 'y,' and calculates a value based on a mathematical expression. The Hill Climbing algorithm starts from a random initial point and iteratively explores its neighbors to find a local maximum. The New Hill Climbing algorithm introduces a modification by selecting one of the best neighbors with higher objective values, even if they don't strictly improve over the current point.

2. Objective Function:
The objective function is defined as follows:

```
def objective_function(x, y):
    return (1 * 1.38 * pow(10, -23) * x) / (1.6 * pow(10, -19)) * (math.log(2.54 / 1e-11) + math.log(y))
```

The function takes two input parameters, 'x' and 'y,' and calculates a value based on a mathematical expression involving arithmetic operations and the natural logarithm. The goal of both Hill Climbing and New Hill Climbing algorithms is to find the values of 'x' and 'y' that maximize this function.

3. Hill Climbing Algorithm:
The Hill Climbing algorithm is a local search technique that starts from a random point and moves iteratively to a neighboring point with a higher objective value.

4. New Hill Climbing Algorithm:
The New Hill Climbing algorithm introduces a modification to the Hill Climbing algorithm by selecting one of the best neighbors with higher objective values, even if they don't strictly improve over the current point. The example demonstrates the usage of both Hill Climbing and New Hill Climbing algorithms to maximize the objective function. The results, i.e., the best solution and the

corresponding objective value, are printed to the console. Additionally, the algorithm's progress is visualized using matplotlib to plot the output voltage per cell against the temperature for each iteration.

5. Conclusion (continued):

5.1. Hill Climbing:
The Hill Climbing algorithm explores the search space by iteratively moving to neighboring points with higher objective values. It starts from a random initial point and continues for a fixed number of iterations. The algorithm does not always guarantee a global optimal solution as it can get stuck in local optima. In this implementation, the step size for exploring neighboring points is fixed, which may affect the algorithm's convergence rate and efficiency.

5.2. New Hill Climbing:
The New Hill Climbing algorithm introduces a modification to the Hill Climbing algorithm by selecting one of the best neighbors with higher objective values, even if they don't strictly improve over the current point. This modification allows the algorithm to escape local optima more effectively and explore a wider area of the search space. However, it is still a local search algorithm and does not guarantee a global optimal solution.

6. Recommendations:
Both Hill Climbing and New Hill Climbing are local search algorithms, which means they may not always find the global optimal solution. To improve the exploration of the search space and increase the likelihood of finding a global optimum, the following recommendations are suggested:

6.1. Exploration Strategy:
Implementing a more sophisticated exploration strategy can help improve the performance of the algorithms. For example, using adaptive step sizes, or exploring more diverse neighborhoods, like employing simulated annealing or genetic algorithms, may increase the chances of finding better solutions.

6.2. Multiple Runs and Random Restarts:

Since both algorithms depend on the initial starting point, running them multiple times with different starting points and selecting the best result can help mitigate the risk of getting trapped in local optima.

6.3. Metaheuristic Algorithms:
Considering more advanced optimization techniques like metaheuristic algorithms (e.g., Genetic Algorithms, Particle Swarm Optimization) can provide a more robust approach to the problem and improve the likelihood of finding global optima.

6.4. Fine-tuning Parameters:
Tuning the parameters such as step sizes and the number of iterations for both algorithms can significantly impact their performance. Experimenting with different parameter values may lead to better results.

7. Conclusion:
In this report, we discussed and analyzed the Hill Climbing and New Hill Climbing algorithms for optimizing an objective function. Both algorithms are local search techniques and have their limitations in finding the global optimal solution. While the New Hill Climbing algorithm introduces a modification to escape local optima better, more advanced optimization techniques may be required to achieve better performance in complex and high-dimensional search spaces. The selection of an appropriate optimization method depends on the problem's characteristics and the trade-off between computational cost and solution quality. Further exploration and experimentation with different optimization strategies may lead to improved results for this particular optimization problem.