# Project Based Learning Report

on

Implementations of the Search Techniques in AI : Generate-And-Test and Hill Climbing algorithms in Python.

Submitted in the partial fulfillment of the requirements

For the Project based learning in AI and Data Mining

in

Electronics & Communication

By

## PRN-2114110433 ANSHUMAN

## PRN-2114110490 UTKARSH MAURYA

**PRN-2214110681 RAM PRAKASH**

Under the guidance of Course In charge Prof. S.R Madkar

Department of Electronic & Communication Engineering

Bharati Vidyapeeth, (Deemed to be University)

College of Engineering, Pune-411043

Academic Year: - 2024-25

## Bharati Vidyapeeth (Deemed to be University) College of Engineering,

**Pune – 411043**

**DEPARTMENT OF ELECTRONICS & COMMUNICATION ENGINEERING**

**CERTIFICATE**

## Certified that the Project Based Learning report entitled, “Implementations of the Search Techniques in AI : Generate-And-Test and Hill Climbing algorithms in Python” is work done by PRN-2114110433 ANSHUMAN

## PRN-2114110436 UTKARSH MAURYA

## PRN-2214110681 RAM PRAKASH

in partial fulfillment of the requirement for the award of credits for Project Based Learning (PBL) in **“AI and Data Mining”,** of Bachelor of Technology semester VII, in Electronics and Communication Engineering.

Date:

Prof. S.R Madkar Dr. Arundhati Shinde

Course In-charge Head of Department – ECE

# INDEX

|  |  |  |
| --- | --- | --- |
| **S.NO**. | **CONTENT** | **PAGE NO.** |
| 1 | PROBLEM STATEMENT | 1 |
| 2 | INTRODUCTION | 2 |
| 3 | SOFTWARE USED | 6 |
| 4 | PROJECT DESCRIPTION | 7 |
| 5 | PROGRAM | 10 |
| 6 | FLOWCHART | 15 |
| 7 | COURSE OUTCOME | 21 |
| 8 | CONCLUSION | 22 |
| 9 | REFERENCE | 25 |

**PROBLEM STATEMENT**

Implementations of the Search Techniques in AI : Generate-And-Test and Hill Climbing algorithms in Python.

The field of Artificial Intelligence (AI) involves various search techniques to find solutions in complex problem spaces. Two such fundamental search techniques are the **Generate-and-Test algorithm** and the **Hill Climbing algorithm**. Both are local search strategies that aim to explore potential solutions and find an optimal or near-optimal solution by evaluating the current state and its neighboring states.

This project focuses on implementing these two algorithms in Python to solve optimization problems. The problem is to:

1. **Understand and implement the Generate-and-Test search algorithm**:
   * This technique involves generating potential solutions randomly and testing them against a goal criterion to find a match. It is a brute-force technique often applied to problems with large or infinite solution spaces.
2. **Understand and implement the Hill Climbing search algorithm**:
   * Hill Climbing is an iterative improvement algorithm that starts with a random solution and tries to improve the solution incrementally by moving toward a better state based on a defined heuristic or fitness function. The algorithm terminates when it reaches a peak, where no further improvements can be made, or when the goal is achieved.

**INTRODUCTION**

Artificial Intelligence (AI) has brought about significant advancements in solving complex problems by mimicking human decision-making processes. One of the core aspects of AI is the ability to search through vast spaces of possible solutions to find the most optimal or satisfactory one. Search algorithms play a crucial role in navigating these problem spaces efficiently. Among various search techniques, **Generate-and-Test** and **Hill Climbing** are two fundamental algorithms often used in AI applications.

**Generate-and-Test** is a simple search technique that works by generating random solutions and testing each against a defined goal until the desired result is found. Though effective in certain scenarios, it can be inefficient due to its brute-force nature, especially in larger or more complex search spaces.

On the other hand, **Hill Climbing** is a more refined local search algorithm. It starts with a random solution and iteratively improves it by exploring neighboring states and moving in the direction of increasing utility or a better solution. This method is more efficient in many cases but can suffer from problems like getting stuck in local optima or plateaus.

In this project, we aim to implement both the Generate-and-Test and Hill Climbing algorithms in Python to solve optimization problems and compare their performance. Through this project, we will explore the strengths and limitations of each algorithm, providing practical insights into their application in real-world scenarios. The project will also demonstrate how AI search techniques can be used for problem-solving and optimization tasks across various domains.

**Generate and Test Search**

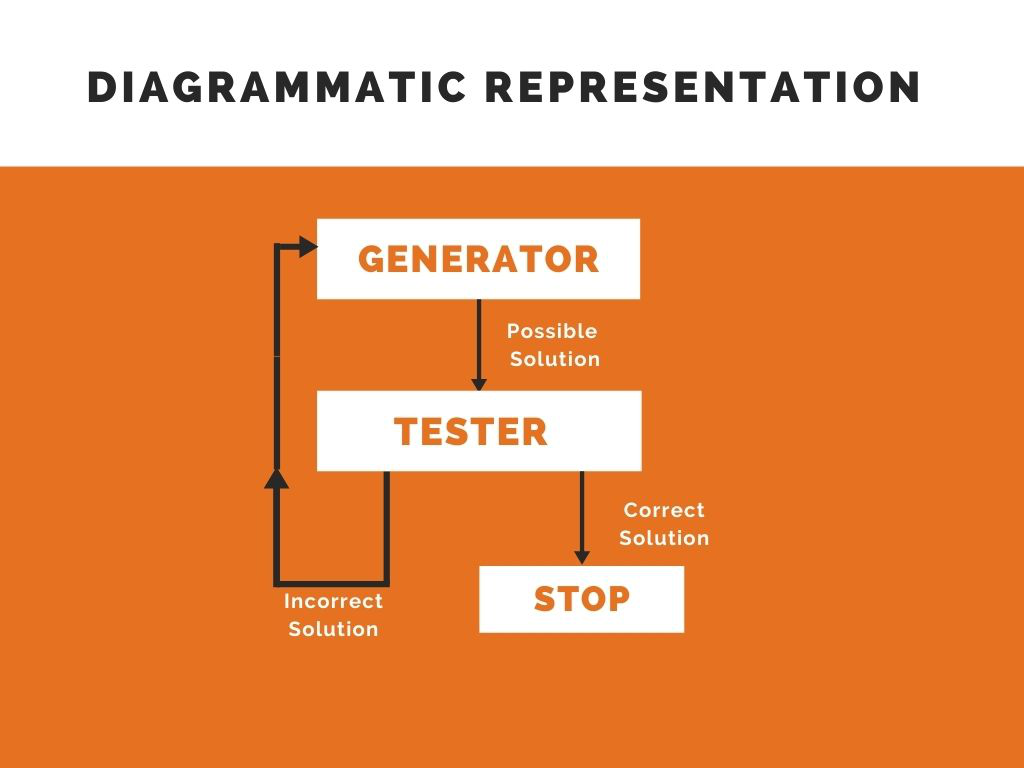
Generate and Test Search is a heuristic search technique based on Depth First Search with Backtracking which guarantees to find a solution if done systematically and there exists a solution. In this technique, all the solutions are generated and tested for the best solution. It ensures that the best solution is checked against all possible generated solutions.

 It is also known as British Museum Search Algorithm as it’s like looking for an exhibit at random or finding an object in the British Museum by wandering randomly.

The evaluation is carried out by the heuristic function as all the solutions are generated systematically in generate and test algorithm but if there are some paths which are most unlikely to lead us to result then they are not considered. The heuristic does this by ranking all the alternatives and is often effective in doing so. Systematic Generate and Test may prove to be ineffective while solving complex problems. But there is a technique to improve in complex cases as well by combining generate and test search with other techniques so as to reduce the search space. For example in Artificial Intelligence Program DENDRAL we make use of two techniques, the first one is Constraint Satisfaction Techniques followed by Generate and Test Procedure to work on reduced search space i.e. yield an effective result by working on a lesser number of lists generated in the very first step.

**Algorithm**

1. Generate a possible solution. For example, generating a particular point in the problem space or generating a path for a start state.
2. Test to see if this is a actual solution by comparing the chosen point or the endpoint of the chosen path to the set of acceptable goal states
3. If a solution is found, quit. Otherwise go to Step 1

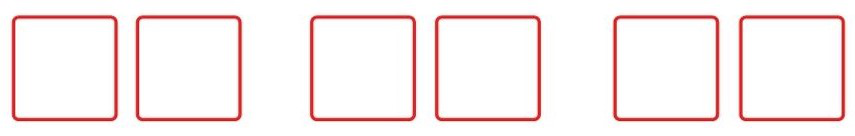


**Properties of Good Generators:**

The good generators need to have the following properties:

* Complete:  Good Generators need to be complete i.e. they should generate all the possible solutions and cover all the possible states. In this way, we can guaranty our algorithm to converge to the correct solution at some point in time.
* Non Redundant:  Good Generators should not yield a duplicate solution at any point of time as it reduces the efficiency of algorithm thereby increasing the time of search and making the time complexity exponential. In fact, it is often said that if solutions appear several times in the depth-first search then it is better to modify the procedure to traverse a graph rather than a tree.
* Informed: Good Generators have the knowledge about the search space which they maintain in the form of an array of knowledge. This can be used to search how far the agent is from the goal, calculate the path cost and even find a way to reach the goal.

Let us take a simple example to understand the importance of a good generator. Consider a pin made up of three 2 digit numbers i.e. the numbers are of the form,



 In this case, one way to find the required pin is to generate all the solutions in a brute force manner for example,

A number in a box

Description automatically generated with medium confidence

The total number of solutions in this case is (100)3 which is approximately 1M. So if we do not make use of any informed search technique then it results in exponential time complexity. Now let’s say if we generate 5 solutions every minute. Then the total numbers generated in 1 hour are 5\*60=300 and the total number of solutions to be generated are 1M. Let us consider the brute force search technique for example linear search whose average time complexity is N/2. Then on an average, the total number of the solutions to be generated are approximately 5 lakhs. Using this technique even if you work for about 24 hrs a day then also you will need 10 weeks to complete the task.

 Now consider using heuristic function where we have domain knowledge that every number is a prime number between 0-99 then the possible number of solutions are (25)3 which is approximately 15,000. Now consider the same case that you are generating 5 solutions every minute and working for 24 hrs then you can find the solution in less than 2 days which was being done in 10 weeks in the case of uninformed search.

We can conclude for here that if we can find a good heuristic then time complexity can be reduced gradually. But in the worst-case time and space complexity will be exponential. It all depends on the generator i.e. better the generator lesser is the time complexity.

**Hill Climbing Algorithm**

In the field of artificial intelligence, the heuristic search algorithm known as "hill climbing" is employed to address optimization-related issues. The algorithm begins in a suboptimal state and incrementally improves it until a predetermined condition is satisfied. The empirical function serves as the basis for the required condition. The algorithm's goal is to get to an improved state called the optimal state from the current state. It attempts to continuously iterate (climb) until it achieves the peak value; thus, the name "Hill Climbing Algorithm" refers to the starting position, which is the non-optimal condition.

On difficult optimization issues, local search techniques are employed to identify a candidate solution that maximizes the criterion. The set of all potential solutions inside the whole functional zone of a problem is referred to as a candidate solution. If you want to learn more about Artificial Intelligence and Machine Learning, [Data Science Certification programs](https://www.knowledgehut.com/data-science-courses) will help you advance your profession.

**What is a Hill Climbing Algorithm?**

**A graph of a graph with orange text

Description automatically generated with medium confidence**

* To discover the mountain's peak or the best solution to the problem, the hill climbing algorithm is a local search algorithm continuously advancing in the direction of increasing elevation or value. When it reaches a peak value where none of its neighbors have a greater value, it ends.
* The hill climbing algorithm is a method for solving mathematical optimization issues. Traveling-salesman is one of the most cited instances of a hill-climbing algorithm. The problem where we need to cut down on the salesman's journey distance.
* Because it just searches inside its good immediate neighbor state and not further afield, it is also known as greedy local search.
* State and value make up the two components of a hill-climbing algorithm node.

Large computational problems can be solved memory-effectively by using the hill climbing algorithm. It considers both the current state and the state immediately nearby. When we wish to optimize or decrease a certain function dependent on the input it is receiving, the hill climbing problem in artificial intelligence is extremely helpful.

The "Traveling Salesman" Problem, in which we must reduce the salesman's journey distance, is the most popular hill climbing algorithm example in AI. Hill Climbing Algorithm is adept at efficiently locating local minima/maxima but may not discover the global optimal (best possible) solution.

Hill climbing is a heuristic strategy, or to put it another way. It is a search technique or informed search technique that assigns various weights based on actual numbers to distinct nodes, branches, and destinations in a path. The search can now be improved using these statistics and the heuristic established in the hill climbing search in the AI model. The hill-climbing algorithm's key characteristics are its high input efficiency and superior heuristic assignment.

**How Does Hill Climbing Algorithm Work?**

The following steps are used by this algorithm to determine the best answer:

* It tries to characterize the present situation as the starting point or initial state.
* It searches for an ideal solution while generalizing the solution to the existing condition. The chosen answer might not be the ideal one.
* It evaluates the generated solution in relation to the goal state, also known as the final state.
* It will determine if the desired state has been attained or not. If this goal is not met, it will look for an alternative approach.

**Features of Hill Climbing**

1. **Generate and Test variant:**The Generate and Test method has an extension called Hill Climbing. Feedback from the Generate and Test approach aids in choosing which way to move through the search space.
2. **Greedy approach:**The hill climbing in artificial intelligence in state space advances in the direction that best optimizes the output taken out in the solution-focused direction. It moves to the end to arrive at the solution while optimizing the cost of function.
3. **No backtracking:**Backtracking to the prior state is not feasible since it cannot remember the system's previous state.
4. **Feedback mechanism:**The program contains a feedback system that aids in choosing the movement's direction. The generate-and-test technique improves the feedback system.
5. **Incremental change:**The algorithm makes small adjustments to the current solution.

**Types of Hill Climbing**

Following are the types of hill climbing in artificial intelligence:

**1. Simple Hill Climbing**

One of the simplest approaches is straightforward hill climbing. It carries out an evaluation by examining each neighbor node's state one at a time, considering the current cost, and announcing its current state. It seeks to find out how the following neighboring state is doing. It attempts to move if the success rate is higher than the current condition; otherwise, it remains in place. Although it is advantageous since it takes less time, the local optima have an impact on it. Therefore, it cannot always guarantee the best optimal solution.

**Algorithm for Simple Hill climbing:**

* Analyze the starting situation. Stop and return success if it's a goal state. If not, the initial state should be set as the current state.
* Continue iterating until the solution state is reached or until no new operators are available to be applied to the current state.
* Choose a state that hasn't yet been applied to the existing state, then do so to create a new state.
* To assess the new state, carry out these.
* Stop and return success if the current state is a goal state.
* If it is superior to the current situation, make it the situation and go on.
* Continue in the loop until a solution is found if it is not an improvement over the situation as it is.
* Exit.

**2. Steepest-Ascent Hill Climbing**

A variant of the straightforward hill-climbing algorithm is the steepest-Ascent algorithm. This method looks at every node that borders the current state and chooses the one that is most near the goal state. This algorithm takes longer since it looks for more neighbors.

**Algorithm for Steepest Ascent Hill climbing:**

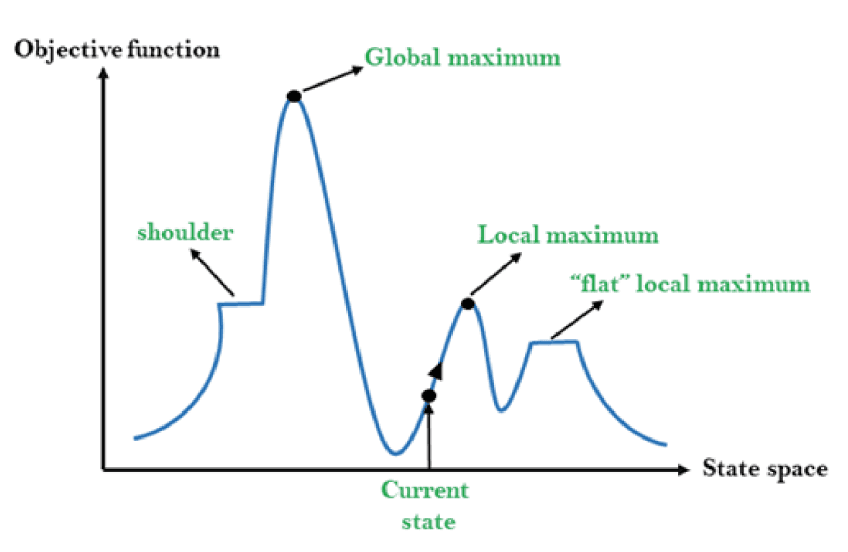
* Analyze the starting situation. Stop and return success if it's a goal state. If not, the initial state should be set as the current state.
* Follow these instructions again and again until a solution is found, or the situation stays the same.
* Choose a state that hasn't yet been used to modify the existing state.
* Create a new "best state" that is initially equivalent to the existing state and then apply it to create the new state.
* Execute these to assess the new state.
* Stop and return success if the current state is a goal state.
* If it is superior to the best state, make it the best state; otherwise, keep going by adding another new state to the loop.
* Set the ideal situation as the current situation.
* Exit.

**3. Stochastic hill climbing**

It is the exact opposite of the methods that were previously explained. With this method, the agent doesn't look up the values of nearby nodes. The agent chooses a neighboring node entirely at random, moves to that node, and then determines whether to continue this path based on the heuristic of that node.

If you want to dive deeper into hill climbing algorithms in artificial intelligence and want to know how much time it takes to get certified as a data scientist, please refer [best Artificial Intelligence Certification](https://www.knowledgehut.com/artificial-intelligence-courses).

**State-space Diagram for Hill Climbing and Analysis**

****

The optimization function and states are graphically represented in a state-space diagram. The local maximum and global maximum are what we seek to establish if the y-axis is the objective function.

The local minimum and the global minimum are what we seek to determine if the cost function reflects this axis. Here you can find more details regarding local minimum, local maximum, global minimum, and global maximum. A straightforward state-space diagram is shown in the diagram below. The state-space is represented by the x-axis, and the objective function has been plotted on the y-axis.

**Different Regions in the State Space Diagram**

1. **Local Maximum:** As the diagram makes clear, this is the state that is marginally superior to its neighboring states but never higher than the highest state.
2. **Global maximum:** Its cost function value is at its highest, and it is the highest state in the state space.
3. **Current State:** This is the condition in which an active agent is present.
4. **Flat local maximums** are what happens when all the neighboring states have the same value and can be visualized as flat spaces (as shown in the diagram).
5. **Shoulder region:** A region with an upward edge, it is also one of the issues with algorithms for climbing hills.

**Problems in Different Regions in Hill climbing**

**1. Local maximum**

All nearby states have a value that is worse than the present state when it reaches its local maximum. Since hill climbing search employs a greedy strategy, it won't progress to a worse state and end itself. Even though there might be a better way, the process will come to an end.

To get around the local maximum issue: Use the backtracking strategy. Keep track of the states you've visited. The search can go back to its initial setup and try a different route if it reaches an unpleasant condition.

**2. Plateau**

All neighbors have the same value on the plateau. Therefore, choosing the ideal course is impossible.

To overcome plateaus: Break through plateaus by taking a huge leap. Choose a state that is far from the one you are in at random.

**3. Ridge**

Any point on a ridge can appear as a peak since all directions of movement are downhill. As a result, the algorithm terminates in this condition.

To get over a Ridge: follow two or more rules before being tested. It suggests acting simultaneously in numerous directions.

**Real-Life Examples**

Here are some examples of real-world applications where Generate-And-Test and Hill Climbing algorithms are used:

**Generate-And-Test Algorithm**

1. **Scheduling**: Generate-And-Test is often used in scheduling problems, such as scheduling tasks on a manufacturing production line or scheduling flights for an airline. The algorithm generates a random schedule and tests it to see if it meets the constraints and objectives.
2. **Resource Allocation**: Generate-And-Test is used in resource allocation problems, such as allocating resources to tasks or projects. The algorithm generates a random allocation and tests it to see if it meets the constraints and objectives.
3. **Combinatorial Optimization**: Generate-And-Test is used in combinatorial optimization problems, such as the traveling salesman problem or the knapsack problem. The algorithm generates a random solution and tests it to see if it is optimal.

**Hill Climbing Algorithm**

1. **Optimization**: Hill Climbing is often used in optimization problems, such as finding the minimum or maximum of a function. The algorithm starts with an initial solution and iteratively applies small changes to the solution, evaluating the new solution and accepting it if it is better than the current solution.
2. **Scheduling**: Hill Climbing is used in scheduling problems, such as scheduling tasks on a manufacturing production line or scheduling flights for an airline. The algorithm starts with an initial schedule and iteratively applies small changes to the schedule, evaluating the new schedule and accepting it if it is better than the current schedule.
3. **Machine Learning**: Hill Climbing is used in machine learning, such as in neural networks or decision trees. The algorithm starts with an initial model and iteratively applies small changes to the model, evaluating the new model and accepting it if it is better than the current model.

**Advantages and Limitations**

**Generate-And-Test Algorithm**

Advantages:

1. Simple to implement
2. Can be effective for small problem sizes
3. Does not require a heuristic function

Limitations:

1. Can be inefficient for large problem sizes
2. May not find the optimal solution
3. Can get stuck in an infinite loop if the solution space is large

**Hill Climbing Algorithm**

Advantages:

1. Can be effective for large problem sizes
2. Can find the optimal solution if the heuristic function is well-designed
3. Does not require a complete enumeration of the solution space

Limitations:

1. Requires a heuristic function, which can be difficult to design
2. Can get stuck in local optima if the heuristic function is not well-designed
3. May not be effective for problems with multiple local optima

In general, Generate-And-Test is a simpler algorithm that is effective for small problem sizes, while Hill Climbing is a more complex algorithm that is effective for larger problem sizes. However, Hill Climbing requires a well-designed heuristic function, which can be difficult to design. Ultimately, the choice of algorithm depends on the specific problem and the characteristics of the solution space.

**SOFTWARE USED**

Python is a high-level, interpreted programming language known for its simplicity, readability, and flexibility. Developed by Guido van Rossum and first released in 1991, Python has since grown into one of the most widely used programming languages across various domains, including web development, data science, artificial intelligence (AI), machine learning (ML), automation, and more.

Python's design philosophy emphasizes code readability with its use of significant whitespace, making it easier to learn and write compared to other languages like C++ or Java. Its syntax allows developers to express complex ideas in fewer lines of code, which enhances productivity and reduces the risk of errors.

Some of Python’s key features include:

1. Easy-to-learn and Use: Python has a clean and simple syntax, making it an ideal choice for both beginners and experienced developers.
2. Interpreted Language: Python code is executed line by line, which allows for quick debugging and dynamic testing during development.
3. Cross-Platform Compatibility: Python can run on various operating systems like Windows, macOS, and Linux without modification.
4. Extensive Libraries and Frameworks: Python boasts a vast standard library and thousands of third-party packages that support a wide range of functionalities, from scientific computing with libraries like NumPy and SciPy, to web development with frameworks like Django and Flask.
5. Community Support: Python has a large and active community that continuously contributes to its development, making it a language with extensive documentation, support, and resources.

In the realm of Artificial Intelligence (AI), Python is particularly powerful due to its extensive libraries like TensorFlow, PyTorch, scikit-learn, and Keras, which simplify the development of AI models and machine learning algorithms. Its simplicity and flexibility make Python a preferred choice for implementing AI search techniques like Generate-and-Test and Hill Climbing.

Python is an open-source, general-purpose programming language that emphasizes code readability and conciseness, making it one of the most accessible and versatile languages for developers. It supports multiple programming paradigms, including procedural, object-oriented, and functional programming, which allows developers to approach problems using different methods.

**PROGRAM**

**Generate-And-Test Algorithm**

import random

def generate\_and\_test(problem, max\_attempts=1000):

  """

  Generate-And-Test algorithm to find a solution to a problem.

  Args:

    problem (function): A function that takes a solution as input and returns True if it's a valid solution, False otherwise.

    max\_attempts (int, optional): Maximum number of attempts to generate a solution. Defaults to 1000.

  Returns:

    solution (any): A valid solution to the problem, or None if no solution is found.

  """

  for \_ in range(max\_attempts):

    solution = generate\_solution(problem)

    if problem(solution):

       return solution

  return None

def generate\_solution(problem):

  """

  Generate a random solution to the problem.

  Args:

    problem (function): A function that takes a solution as input and returns True if it's a valid solution, False otherwise.

  Returns:

    solution (any): A random solution to the problem.

  """

  # This function should generate a random solution to the problem

  # For example, if the problem is to find a number between 1 and 100, this function could return a random integer between 1 and 100

  pass

**Flow Chart**

1. Start
2. Generate a random solution to the problem
3. Test the solution to see if it's valid
4. If the solution is valid, return it
5. If the solution is not valid, repeat steps 2-4 until a valid solution is found or the maximum number of attempts is reached
6. If no valid solution is found, return None

**Hill Climbing Algorithm**

import random

def hill\_climbing(problem, max\_iterations=1000):

  """

  Hill Climbing algorithm to find a solution to a problem.

  Args:

    problem (function): A function that takes a solution as input and returns a score indicating how good the solution is.

    max\_iterations (int, optional): Maximum number of iterations to perform. Defaults to 1000.

  Returns:

    solution (any): A solution to the problem, or None if no solution is found.

  """

  current\_solution = generate\_initial\_solution(problem)

  current\_score = problem(current\_solution)

  for \_ in range(max\_iterations):

    neighbor = generate\_neighbor(current\_solution)

    neighbor\_score = problem(neighbor)

    if neighbor\_score > current\_score:

       current\_solution = neighbor

       current\_score = neighbor\_score

  return current\_solution

def generate\_initial\_solution(problem):

  """

  Generate an initial solution to the problem.

  Args:

    problem (function): A function that takes a solution as input and returns a score indicating how good the solution is.

  Returns:

    solution (any): An initial solution to the problem.

  """

  # This function should generate an initial solution to the problem

  # For example, if the problem is to find a number between 1 and 100, this function could return a random integer between 1 and 100

  pass

def generate\_neighbor(solution):

  """

  Generate a neighbor of the current solution.

  Args:

    solution (any): The current solution.

  Returns:

    neighbor (any): A neighbor of the current solution.

  """

  # This function should generate a neighbor of the current solution

  # For example, if the problem is to find a number between 1 and 100, this function could return a number that is one more or one less than the current solution

  pass

**Flow Chart**

1. Start
2. Generate an initial solution to the problem
3. Evaluate the initial solution using the problem function
4. Generate a neighbor of the current solution
5. Evaluate the neighbor using the problem function
6. If the neighbor is better than the current solution, update the current solution and repeat steps 4-6
7. If the neighbor is not better than the current solution, repeat steps 4-6 until a better solution is found or the maximum number of iterations is reached
8. Return the final solution

**Problem Statement**

We have a set of tasks, each with a processing time and a deadline. We want to schedule these tasks on a single machine to minimize the total processing time, while ensuring that each task is completed before its deadline.

**Generate-And-Test Algorithm**

import random

def generate\_and\_test(tasks, max\_attempts=1000):

  """

  Generate-And-Test algorithm to schedule tasks on a single machine.

  Args:

    tasks (list): A list of tasks, each represented as a tuple of (processing\_time, deadline).

    max\_attempts (int, optional): Maximum number of attempts to generate a schedule. Defaults to 1000.

  Returns:

    schedule (list): A list of tasks in the order they should be scheduled, or None if no schedule is found.

  """

  for \_ in range(max\_attempts):

    schedule = generate\_schedule(tasks)

    if is\_valid\_schedule(schedule, tasks):

       return schedule

  return None

def generate\_schedule(tasks):

  """

  Generate a random schedule for the tasks.

  Args:

    tasks (list): A list of tasks, each represented as a tuple of (processing\_time, deadline).

  Returns:

    schedule (list): A list of tasks in a random order.

  """

  random.shuffle(tasks)

  return tasks

def is\_valid\_schedule(schedule, tasks):

  """

  Check if a schedule is valid.

  Args:

    schedule (list): A list of tasks in the order they should be scheduled.

    tasks (list): A list of tasks, each represented as a tuple of (processing\_time, deadline).

  Returns:

    bool: True if the schedule is valid, False otherwise.

  """

  current\_time = 0

  for task in schedule:

    processing\_time, deadline = task

    if current\_time + processing\_time > deadline:

       return False

    current\_time += processing\_time

  return True

# Example usage

tasks = [(10, 20), (5, 15), (8, 18), (12, 22)]

schedule = generate\_and\_test(tasks)

if schedule:

  print("Schedule:", schedule)

  print("Total processing time:", sum(task[0] for task in schedule))

else:

  print("No schedule found")

**Hill Climbing Algorithm**

import random

def hill\_climbing(tasks, max\_iterations=1000):

  """

  Hill Climbing algorithm to schedule tasks on a single machine.

  Args:

    tasks (list): A list of tasks, each represented as a tuple of (processing\_time, deadline).

    max\_iterations (int, optional): Maximum number of iterations to perform. Defaults to 1000.

  Returns:

    schedule (list): A list of tasks in the order they should be scheduled, or None if no schedule is found.

  """

  current\_schedule = generate\_initial\_schedule(tasks)

  current\_score = evaluate\_schedule(current\_schedule, tasks)

  for \_ in range(max\_iterations):

    neighbor = generate\_neighbor(current\_schedule)

    neighbor\_score = evaluate\_schedule(neighbor, tasks)

    if neighbor\_score < current\_score:

       current\_schedule = neighbor

       current\_score = neighbor\_score

  return current\_schedule

def generate\_initial\_schedule(tasks):

  """

  Generate an initial schedule for the tasks.

  Args:

    tasks (list): A list of tasks, each represented as a tuple of (processing\_time, deadline).

  Returns:

    schedule (list): A list of tasks in a random order.

  """

  random.shuffle(tasks)

  return tasks

def generate\_neighbor(schedule):

  """

  Generate a neighbor of the current schedule.

  Args:

    schedule (list): A list of tasks in the order they should be scheduled.

  Returns:

    neighbor (list): A list of tasks in a new order.

  """

  i, j = random.sample(range(len(schedule)), 2)

  schedule[i], schedule[j] = schedule[j], schedule[i]

  return schedule

def evaluate\_schedule(schedule, tasks):

  """

  Evaluate the quality of a schedule.

  Args:

    schedule (list): A list of tasks in the order they should be scheduled.

    tasks (list): A list of tasks, each represented as a tuple of (processing\_time, deadline).

  Returns:

    score (int): The total processing time of the schedule.

  """

  current\_time = 0

  score = 0

  for task in schedule:

    processing\_time, deadline = task

    if current\_time + processing\_time > deadline:

       score += 1000 # penalty for missing deadline

    current\_time += processing\_time

    score += processing\_time

  return score

# Example usage

tasks = [(10, 20), (5, 15), (8, 18), (12, 22)]

schedule = hill\_climbing(tasks)

if schedule:

  print("Schedule:", schedule)

  print("Total processing time:", sum(task[0] for task in schedule))

else:

  print("No schedule found")

****

In both examples, we define a set of tasks with processing times and deadlines, and use the Generate-And-Test and Hill Climbing algorithms to find a schedule that minimizes the total processing time while ensuring that each task is completed before its deadline. The Hill Climbing algorithm uses a neighbor generation function to iteratively improve the schedule, while the Generate-And-Test algorithm generates a random schedule and checks if it is valid.

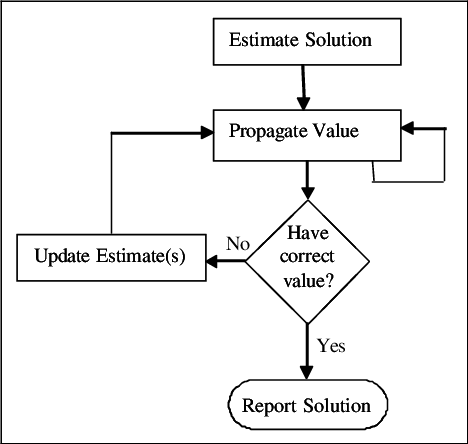
Note that the Hill Climbing algorithm may not always find the optimal solution, as it relies on the quality of the initial schedule and the neighbor generation function. The Generate-And-Test algorithm, on the other hand, is guaranteed to find a valid schedule if one exists, but may take longer to find a solution.

**FLOWCHART**

**Hill Climbing Algorithm**

# 25: Flowchart of Hill climbing algorithm. | Download ...

**Generate-And-Test Algorithm**



**COURSE OUTCOME**

**CO1** Evaluate various problem-solving agents in AI

**CO2** Design and analyse search techniques and game playing techniques

**CONCLUSION**

In this project, we successfully implemented and analyzed two fundamental search techniques in Artificial Intelligence: **Generate-and-Test** and **Hill Climbing algorithms**. These algorithms, while conceptually simple, offer valuable insights into local search strategies and how they navigate large search spaces.

The **Generate-and-Test** algorithm, as expected, demonstrates a brute-force approach by generating potential solutions randomly and testing each one until the goal is found. While easy to implement, it lacks efficiency, especially in large or complex search spaces, due to the absence of any guidance or optimization in the search process.

On the other hand, the **Hill Climbing algorithm** showed more intelligent behavior by iteratively improving a candidate solution based on neighboring states. Although Hill Climbing provides a more directed search by evaluating the fitness of neighboring solutions, it has its limitations. Specifically, it is prone to getting stuck in local optima, where it can no longer improve the solution, even though better solutions may exist elsewhere in the search space.

Through various test cases and scenarios implemented in Python, the algorithms' strengths and weaknesses became evident. The **Generate-and-Test** algorithm excels in its simplicity but is inefficient for large-scale problems. Conversely, the **Hill Climbing** algorithm offers efficiency but can falter in more complex problem spaces due to the risk of reaching local optima.

The project achieved the following outcomes:

* A clear understanding of the workings of **Generate-and-Test** and **Hill Climbing** search techniques.
* Successful implementation of both algorithms in Python, demonstrating their behavior with practical test cases.
* Analysis of the performance and limitations of each algorithm in solving optimization problems.
* Insight into the importance of search techniques in AI and how they can be applied to real-world scenarios that require exploring large solution spaces.

This project laid the foundation for understanding more advanced search algorithms and optimization techniques that could mitigate the limitations faced by these basic algorithms. It opens the door for further exploration into search strategies like **Simulated Annealing**, **Genetic Algorithms**, or *A Search*\*, which offer improved efficiency and robustness in navigating complex problem spaces.

**REFERENCES**.

1. **Russell, S. J., & Norvig, P. (2020)**. *Artificial Intelligence: A Modern Approach* (4th ed.). Pearson Education.

This textbook is a foundational reference for AI, covering various search algorithms, including Generate-and-Test and Hill Climbing, with theoretical explanations and examples.

1. **Python Documentation**. (2023). *Python Software Foundation*. Retrieved from <https://docs.python.org/3/>
   * The official documentation for Python, which provides comprehensive guidance on Python libraries and functions necessary for implementing AI algorithms.
2. **Mitchell, M. (1997)**. *An Introduction to Genetic Algorithms*. MIT Press.

Though focused on genetic algorithms, this book provides insights into optimization and search techniques like Hill Climbing, offering a broader context for heuristic methods.