**AI with Search Techniques and Games**



**WELCOME INTELLIGENT SYSTEMS STUDENTS!**

**I. INTRODUCTION:**

We begin this module by developing a motivation for learning about the Artificial Intelligence with search techniques and games. In this module, you will learn the objective of Heuristic Search, Hill Climbing in AI, Constraint Satisfaction Problems, Simulated Annealing Heuristic Search and Best First Search.

**II. OBJECTIVES:**

After completing this course, you will be able to learn about:

* Determine the objective of Heuristic Search in AI.
* Identify the concept of Hill Climbing in AI.
* Explain what is Constraint Satisfaction Problems, Simulated Annealing Heuristic Search and Best First Search.

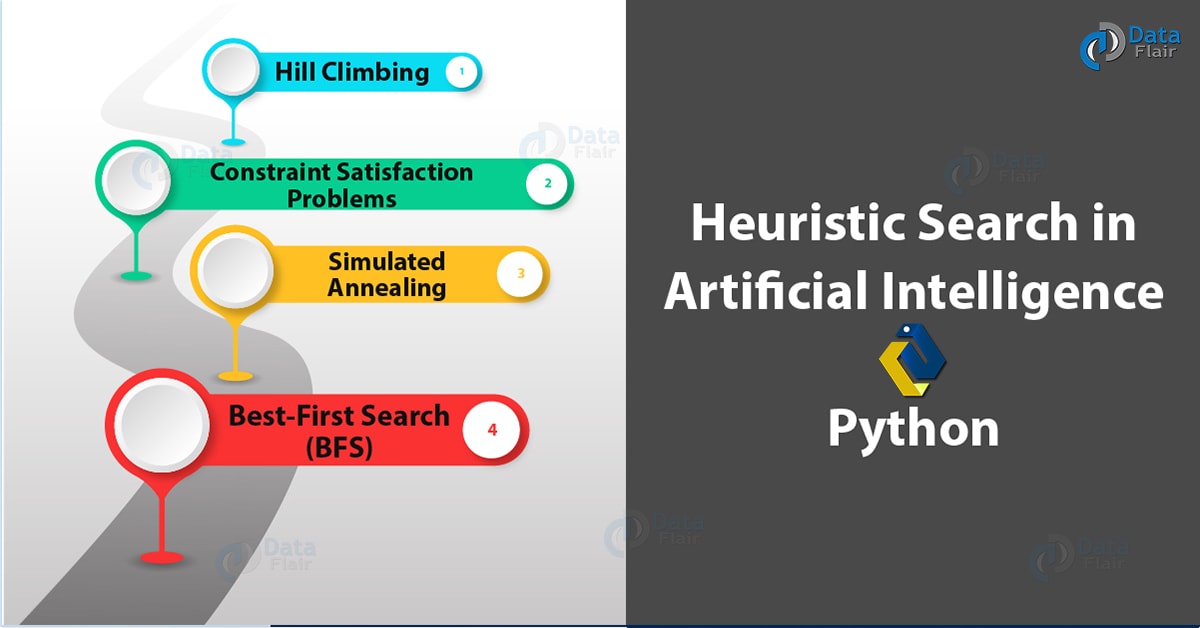
**III. DEVELOPMENT OF THE LESSON**

**COURSE MATERIALS:**

**Heuristics**

## 1. Objective – Heuristic Search

In this [**Python AI tutorial**](https://data-flair.training/blogs/python-ai-tutorial/), we will discuss the rudiments of Heuristic Search, which is an integral part of Artificial Intelligence. We will talk about different techniques like Constraint Satisfaction Problems, Hill Climbing, and Simulated Annealing. Also, we will implement CSP in Python.  
So, let’s begin Heuristic Search in AI Tutorial.



*Heuristic Search in Artificial Intelligence – Python*

[**First, let’s revise the Artificial Intelligence Tutorial**](https://data-flair.training/blogs/artificial-intelligence-introduction/)

## 2. What is a Heuristic Search?

A Heuristic is a technique to solve a problem faster than classic methods, or to find an approximate solution when classic methods cannot. This is a kind of a shortcut as we often trade one of optimality, completeness, accuracy, or precision for speed. A Heuristic (or a heuristic function) takes a look at search algorithms. At each branching step, it evaluates the available information and makes a decision on which branch to follow. It does so by ranking alternatives. The Heuristic is any device that is often effective but will not guarantee work in every case.

[**You must take a look at NLP Tutorial**](https://data-flair.training/blogs/nlp-tutorial-natural-language-processing/)

So why do we need heuristics? One reason is to produce, in a reasonable amount of time, a solution that is good enough for the problem in question. It doesn’t have to be the best- an approximate solution will do since this is fast enough. Most problems are exponential. Heuristic Search let us reduce this to a rather polynomial number. We use this in AI because we can put it to use in situations where we can’t find known algorithms.  
We can say Heuristic Techniques are weak methods because they are vulnerable to combinatorial explosion.

## 3. Heuristic Search Techniques in Artificial Intelligence

Briefly, we can taxonomize such techniques of Heuristic into two categories:



*Heuristic Search Techniques in Artificial Intelligence*

### a. Direct Heuristic Search Techniques in AI

Other names for these are Blind Search, Uninformed Search, and Blind Control Strategy. These aren’t always possible since they demand much time or memory. They search the entire state space for a solution and use an arbitrary ordering of operations. Examples of these are Breadth First Search (BFS) and Depth First Search (DFS).

[**Do you know about NLTK Python**](https://data-flair.training/blogs/nltk-python-tutorial/)

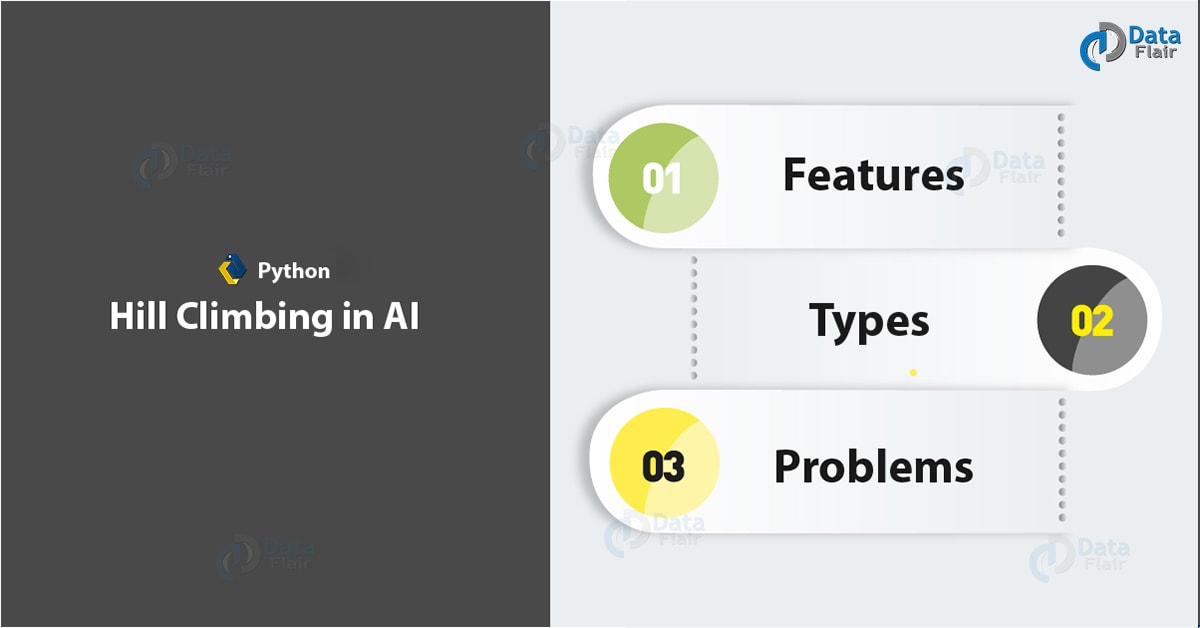
### b. Weak Heuristic Search Techniques in AI

Other names for these are Informed Search, Heuristic Search, and Heuristic Control Strategy. These are effective if applied correctly to the right types of tasks and usually demand domain-specific information. We need this extra information to compute preference among child nodes to explore and expand. Each node has a heuristic function associated with it. Examples are Best First Search (BFS) and A\*.  
Before we move on to describe certain techniques, let’s first take a look at the ones we generally observe. Below, we name a few.

* Best-First Search
* A\* Search
* Bidirectional Search
* Tabu Search
* Beam Search
* Simulated Annealing
* Hill Climbing
* Constraint Satisfaction Problems

## 4. Hill Climbing in Artifical Intelligence

First, let’s talk about Hill Climbing in Artifical Intelligence. This is a heuristic for optimizing problems mathematically. We need to choose values from the input to maximize or minimize a real function. It is okay if the solution isn’t the global optimal maximum.



*Heuristic Search Techniques – Hill Climbing*

[**Let’s discuss Python Speech Recognition**](https://data-flair.training/blogs/python-speech-recognition-ai/)

One such example of Hill Climbing will be the widely discussed Travelling Salesman Problem- one where we must minimize the distance he travels.

### a. Features of Hill Climbing in AI

Let’s discuss some of the features of this algorithm (Hill Climbing):

* It is a variant of the generate-and-test algorithm
* It makes use of the greedy approach

This means it keeps generating possible solutions until it finds the expected solution, and moves only in the direction which optimizes the cost function for it.

### b. Types of Hill Climbing in AI



*Heuristic Search – Types of Hill Climbing in Artifical Intelligence*

* **Simple Hill Climbing-** This examines one neighboring node at a time and selects the first one that optimizes the current cost to be the next node.
* **Steepest Ascent Hill Climbing-** This examines all neighboring nodes and selects the one closest to the solution state.
* **Stochastic Hill Climbing-** This selects a neighboring node at random and decides whether to move to it or examine another.

[**Let’s revise Python Unit testing**](https://data-flair.training/blogs/python-unittest/)

Let’s take a look at the algorithm for simple hill climbing.

1. Evaluate initial state- if goal state, stop and return success. Else, make initial state current.
2. Loop until the solution reached or until no new operators left to apply to current state:

a. Select new operator to apply to the current producing new state.

b. Evaluate new state:

* If a goal state, stop and return success.
* If better than the current state, make it current state, proceed.
* Even if not better than the current state, continue until the solution reached.

1. Exit.

### c. Problems with Hill Climbing in AI

We usually run into one of three issues-

* **Local Maximum-** All neighboring states have values worse than the current. The greedy approach means we won’t be moving to a worse state. This terminates the process even though there may have been a better solution. As a workaround, we use backtracking.
* **Plateau-** All neighbors to it have the same value. This makes it impossible to choose a direction. To avoid this, we randomly make a big jump.
* **Ridge-** At a ridge, movement in all possible directions is downward. This makes it look like a peak and terminates the process. To avoid this, we may use two or more rules before testing.

[**Do you know about Python Assert Statements**](https://data-flair.training/blogs/python-assert/)

## 5. Constraint Satisfaction Problems (CSP)

A constraint is nothing but a limitation or a restriction. [**Working with AI**](https://data-flair.training/blogs/artificial-intelligence-applications/), we may need to satisfy some constraints to solve problems. Let’s try solving a problem this way, shall we?

Let’s talk of a magic square. This is a sequence of numbers- usually integers- arranged in a square grid. The numbers in each row, each column, and each diagonal all add up to a constant which we call the *Magic Constant*. Let’s implement this with Python.

1. def **magic\_square**(matrix):
2. size=**len**(matrix[0])
3. sum\_list=[]
4. for col in **range**(size): #Vertical sum
5. sum\_list.**append**(**sum**(row[col] for row in matrix))
6. sum\_list.**extend**([**sum**(lines) for lines in matrix])#Horizontal sum
7. result1=0
8. for i in **range**(0,size):
9. result1+=matrix[i][i]
10. sum\_list.**append**(result1)
11. result2=0
12. for i in **range**(size-1,-1,-1):
13. result2+=matrix[i][i]
14. sum\_list.**append**(result2)
15. if **len**(**set**(sum\_list))>1:
16. return False
17. return True

Now let’s run this code.

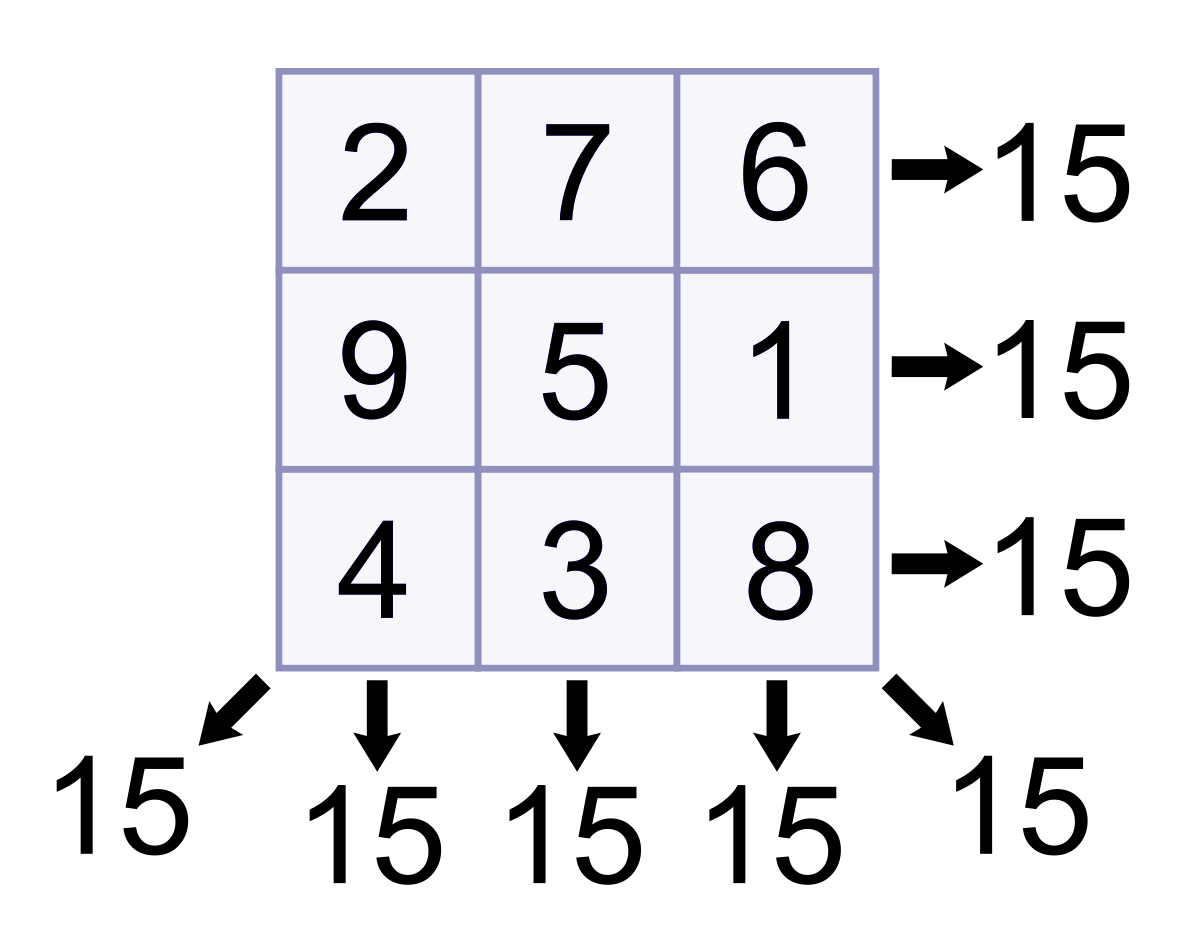
1. >>> **magic\_square**([[1,2,3],[4,5,6],[7,8,9]])

**False**  
This is not a magic square. The numbers in the rows/columns/diagonals do not add up to the same value. Let’s try another list of lists.

[**Have a look at AI Neural Networks**](https://data-flair.training/blogs/neural-network/)

1. >>> **magic\_square**([[2,7,6],[9,5,1],[4,3,8]])

**True**



*Heuristic Search – Magic*

Since the values add up to the constant 15 in all directions, surely, this is a magic square!

## 6. Simulated Annealing Heuristic Search

In metallurgy, when we slow-cool metals to pull them down to a state of low energy gives them exemplary amounts of strength. We call this annealing. While high temperatures observe much random movement, low temperatures notice little randomness.

In AI, we take a cue from this to produce something called simulated annealing. This is a way of optimization where we begin with a random search at a high temperature and reduce the temperature slowly. Eventually, as the temperature approaches zero, the search becomes pure greedy descent. At each step, this processes randomly selects a variable and a value. It accepts the assignment only when it is an improvement or doesn’t lead to more conflict. If not, it checks if the temperature is much worse than the current assignment to accept the assignment with some probability.

[**Do you know about Python ternary Operators**](https://data-flair.training/blogs/python-ternary-operator/)

An annealing schedule defines how the temperature drops as the search progress. A very common schedule is geometric cooling. If we begin with a temperature of 10 and multiply by 0.97 after every step, then after 100 steps, we’re left with a temperature of 0.48.

## 7. Best-First Search (BFS) Heuristic Search

Often dubbed BFS, Best First Search is an informed search that uses an evaluation function to decide which adjacent is the most promising before it can continue to explore. Breadth- and Depth- First Searches blindly explore paths without keeping a cost function in mind. Things aren’t the same with BFS, though. Here, we use a priority queue to store node costs. Let’s understand BFS Heuristic Search through pseudocode.

1. Define list OPEN with single node *s*– the start node.
2. IF list is empty, return failure.
3. Remove node *n* (node with best score) from list, move it to list CLOSED.
4. Expand node *n*.
5. IF any successor to *n* is the goal node, return success and trace path from goal node to *s* to return the solution.
6. FOR each successor node:

* Apply evaluation function *f*.
* IF the node isn’t in either list, add it to list OPEN.

1. Loop to step 2.

**Unit 2: Tic Tac Toe**

To play our game, we will need some functions. The first thing to do is to summarize our needs.

Tic-tac-toe is played on a board. Our first function will be a display\_board function. Then, we need to ask the player if they would like to use either a cross (X) or a circle (O). We’ve got our second function.

We also need the player to place the marker (third function). But before placing it, the user has to choose where to place it (fourth function) and determine whether this location is available (fifth function).

We also want to check each time a marker is placed on whether someone has won (sixth function) or whether the board is full (seventh function).

Finally, we would like a replay function. As we all have time to spend, it can be great if we could play as much as we want without relaunching the program.

The first function will be called display\_board. We will create a blank board that contains numpad numbers. Those numbers will be replaced by either an X or an O after the player chooses.

Players’ choices will be stored in a list of ten items (just because item number 0 cannot be a user’s choice). This list will be defined in the main program, and will look like this: board = [‘#’,‘#’,‘#’,‘#’,‘#’,‘#’,‘#’,‘#’,‘#’,‘#’].

This function takes the board list as a parameter and replaces the case number inside the blankBoard with either the player’s choice or a blank space.

The second function will ask player one which marker they want. Choices can be X/x or O/o. If another choice is made by the user, the program must re-ask for an input.

Here, we simply use a while True statement, which will be exited by a return if the input matches the possibilities. Note that we are applying the input if player1.upper() == ‘X’: to avoid double checks.

The third function will be the place\_marker. This one is pretty small.

def place\_marker(board, marker, position):  
 board[position] = marker  
 return board

We just pass the board list, the marker, and the position chosen. The function will assign the marker to our list, replacing the # character at the given position.

Hereafter, both our functions will ask the player for a position and check whether the chosen position is empty or not. These two functions are linked, and that’s why I put these two behind together.

First, the space\_check function will receive two args, the board list and the position. It will return True if the field number position is equal to #. This means that the space is free. If it’s not equal, this means that this field has been already played, so it returns False.

Next, the player\_choice function takes the board as param. It asks the user for a choice. As long as the chosen place isn’t free, it keeps asking the user for a valid field. Finally, it returns the user’s choice.

Only three functions remain: Is the board full? Has someone won? Do you want to play again?

def full\_board\_check(board):  
 return len([x for x in board if x == '#']) == 1

The question “is the board full?” can be checked with the above code.

We use here a comprehensive list with an if statement. What we do is parse the board list and extract all items matching the hashtag (#) character into a list. Then we calculate the length of this list, and if it’s equal to 1, we return True (i.e., our board is full); if not, we return False.

The next one to check is whether someone did win the game. I’m quite sure it can be improved. We are just checking if we got a line or diagonal with the same marker. The two required parameters are the board list and the mark we want to check.

We’re at our last function before our main program, and that’s just asking the users if they want to play again.

def replay():  
 playAgain = input("Do you want to play again (y/n) ? ")  
 if playAgain.lower() == 'y':  
 return True  
 if playAgain.lower() == 'n':  
 return False

We are done with our functions. Now it’s time to implement all of these into our main program.

We run the program only if it’s run directly. If we load the function from another Python script, what’s below the above if statement will not be executed. To learn more about the builtin variable \_\_name\_\_ you can have a look here for an example: <https://www.guru99.com/learn-python-main-function-with-examples-understand-main.html>

<https://gist.github.com/30d4d2e378f2f0d671ccd59f7a2c99d2>

What do we need here? Maybe a counter we call i. This one will be used to determine which player is playing, using the modulo function: if i % 2 == 0, we know that it’s player two’s turn.

Next, we ask our user to choose the marker they want using the function player\_input. Do not forget this function return two values that we are storing in the players var.

We also need to start with an empty board list, which is defined by the statement board = [‘#’] \* 10.

We are ready to start. We begin with a while True loop, which begins with our check of whether the board is full. If it’s not full, we enter in a second while loop. We can break this down into five blocks:

1. We ask the user for an input:  
   position = player\_choice(board)
2. We determine who is playing:

if i % 2 == 0:  
marker = players[1]  
else: marker = players[0]

3. We place the marker and display the board to the players, and then we increment our counter i to switch player:

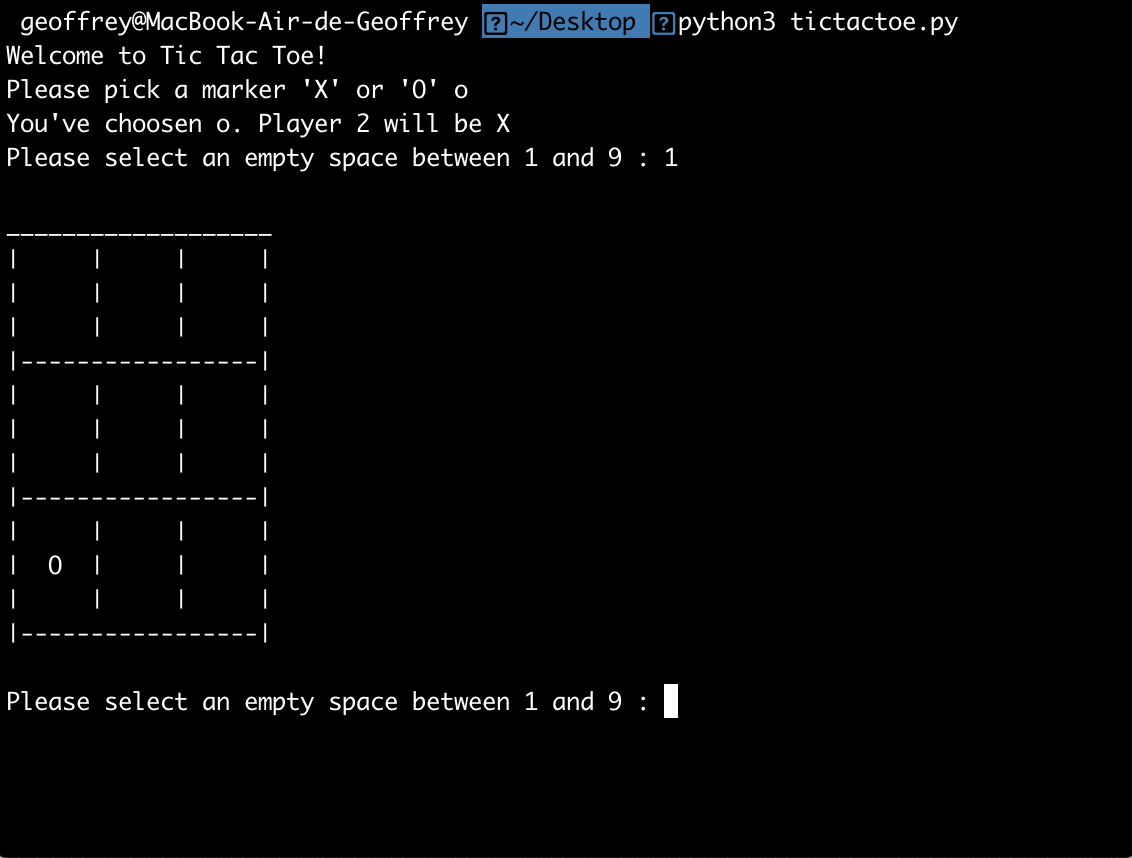
# Play !  
 place\_marker(board, marker, int(position))  
 # Check the board  
 display\_board(board)  
 i += 1

3. We check if someone won:

if win\_check(board, marker):  
 print(“You won !”)  
 break  
 game\_on=full\_board\_check(board)

4. Finally, we ask for replay. If users want to play again, we re-init our board and counter vars, and we go back to our first while True loop.

Now our code is ready to work, and you are ready to play. You just need to paste the full code into a .py file, and run it with the Python command.  
Here is what you will have as output:



TicTacToe game overview

All of this tutorial has been made watching a fantastic Udemy course, [Complete Python Bootcamp](https://www.udemy.com/course/complete-python-bootcamp/) .

You can find the full code here in this GitHub gist:

I hope you enjoyed this tutorial, and that this tic-tac-toe game will amuse



We had just finished the discussion on Principles of AI. Let’s move on to the next higher level of activity/ies or exercise/s that demonstrate your potential skills/knowledge of what you have learned.

**IV. ANALYSIS, APPLICATION AND EXPLORATION**

✂

**ACTIVITY 1**

Name: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_ Year & Section: \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_

**Choose the correct answer:**

|  |  |
| --- | --- |
| **1.** | What is Artificial intelligence?  (a)   Putting your intelligence into Computer  (b)   Programming with your own intelligence  (c)   Making a Machine intelligent  (d)   Playing a Game  (e)   Putting more memory into Computer |
| **2.** | Which is not the commonly used programming language for AI?  (a)  PROLOG           (b)  Java                  (c)  LISP                  (d)  Perl             (e)  Java script. |
| **3.** | What is state space?  (a)   The whole problem  (b)   Your Definition to a problem  (c)   Problem you design  (d)   Representing your problem with variable and parameter  (e)   A space where You know the solution. |
| **4.** | A production rule consists of  (a)  A set of Rule                                    (b)  A sequence of steps  (c)  Both (a) and (b)                                (d)  Arbitrary representation to problem  (e)  Directly getting solution. |
| **5.** | Which search method takes less memory?  (a)  Depth-First Search                            (b)  Breadth-First search  (c)  Both (a) and (b)                                (d)  Linear Search.  (e)  Optimal search. |

**V. GENERALIZATION**

* BFS finds the shortest path to the destination whereas DFS goes to the bottom of a subtree, then backtracks.
* The full form of BFS is Breadth-First Search while the full form of DFS is Depth First Search.
* BFS uses a queue to keep track of the next location to visit. whereas DFS uses a stack to keep track of the next location to visit.
* BFS traverses according to tree level while DFS traverses according to tree depth.
* BFS is implemented using FIFO list on the other hand DFS is implemented using LIFO list.
* In BFS, you can never be trapped into finite loops whereas in DFS you can be trapped into infinite loops.

**VI. REFERENCES:**

* Nagy (2018), Artificial Intelligence and Machine Learning Fundamentals.
* Kim (2017), MATLAB Deep Learning: With Machine Learning, Neural Networks and Artificial Intelligence.
* Geron (2017), Hands-On Machine Learning with Scikit-Learn and TensorFlow: Concepts, Tools, and Techniques.
* <https://www.udemy.com/course/artificial-intelligence-and-machine-learning-fundamentals/?fbclid=IwAR0CCSgtFGTFlqD08s1wZYRSAsLeewOTl8fwUtojODDgTzrqx_KG4WeRcOU>.
* <https://data-flair.training/blogs/heuristic-search-ai/>.

**CONGRATULATIONS** on reaching the end of this module!

You may now proceed to the next module.

Don’t forget to submit all the exercises, activities and portfolio

on \_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_\_.

**KEEP UP THE GOOD WORK**.

Well Done!!!

