Home Assignment

Core Paper: Machine Leaning, NLP and AI

Submitted By: Arvind K Sinha

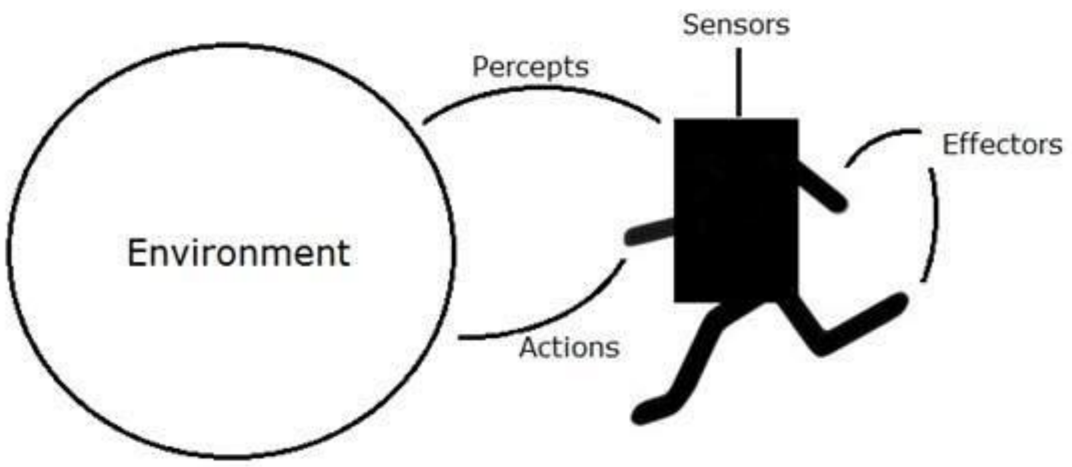
Student Name: Arvind K Sinha

Reg No: 219048101003

Submission Date:9th Oct 2021

An **agent** is anything that can perceive its environment through **sensors** and acts upon that environment through **effectors.**

- A **human agent** has sensory organs such as eyes, ears, nose, tongue and skin parallel to the sensors, and other organs such as hands, legs, mouth, for effectors.
- A **robotic agent** replaces cameras and infrared range finders for the sensors, and various motors and actuators for effectors.
- A **software agent** has encoded bit strings as its programs and actions.



Agent Terminology

- **Performance Measure of Agent** − It is the criteria, which determines how successful an agent is.
- **Behavior of Agent** − It is the action that agent performs after any given sequence of percepts.
- **Percept** − It is agent's perceptual inputs at a given instance.
- **Percept Sequence** − It is the history of all that an agent has perceived till date.
- **Agent Function** − It is a map from the precept sequence to an action.

Rationality

Rationality is nothing but status of being reasonable, sensible, and having good sense of judgment.

Rationality is concerned with expected actions and results depending upon what the agent has perceived. Performing actions with the aim of obtaining useful information is an important part of rationality.

What is Ideal Rational Agent?

An ideal rational agent is the one, which is capable of doing expected actions to maximize its performance measure, on the basis of −

- Its percept sequence
- Its built-in knowledge base

Rationality of an agent depends on the following −

- The **performance measures**, which determine the degree of success.
- Agent's **Percept Sequence** till now.
- The agent's **prior knowledge about the environment**.
- The **actions** that the agent can carry out.

A rational agent always performs right action, where the right action means the action that causes the agent to be most successful in the given percept sequence. The problem the agent solves is characterized by Performance Measure, Environment, Actuators, and Sensors (PEAS).

The Structure of Intelligent Agents

Agent's structure can be viewed as −

- Agent = Architecture + Agent Program
- Architecture = the machinery that an agent executes on.
- Agent Program = an implementation of an agent function.

Simple Reflex Agents

- They choose actions only based on the current percept.
- They are rational only if a correct decision is made only on the basis of current precept.
- Their environment is completely observable.

Condition-Action Rule − It is a rule that maps a state (condition) to an action.

Searching

Uninformed search is a class of general-purpose search algorithms which operates in brute force-way. Uninformed search algorithms do not have additional information about state or search space other than how to traverse the tree, so it is also called blind search.

Following are the various types of uninformed search algorithms:

1. **Breadth-first Search**
2. **Depth-first Search**
3. **Depth-limited Search**
4. **Iterative deepening depth-first search**
5. **Uniform cost search**
6. **Bidirectional Search**

1. Breadth-first Search:

- Breadth-first search is the most common search strategy for traversing a tree or graph. This algorithm searches breadthwise in a tree or graph, so it is called breadth-first search.
- BFS algorithm starts searching from the root node of the tree and expands all successor node at the current level before moving to nodes of next level.
- The breadth-first search algorithm is an example of a general-graph search algorithm.
- Breadth-first search implemented using FIFO queue data structure.

Advantages:

- BFS will provide a solution if any solution exists.
- If there are more than one solutions for a given problem, then BFS will provide the minimal solution which requires the least number of steps.

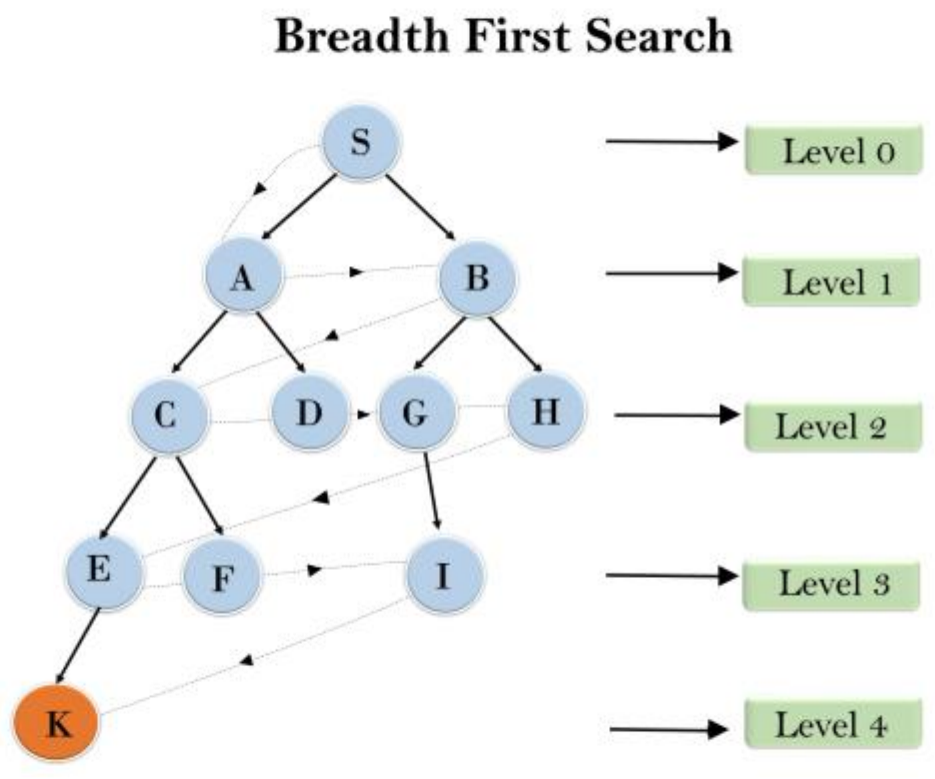
Disadvantages:

- It requires lots of memory since each level of the tree must be saved into memory to expand the next level.
- BFS needs lots of time if the solution is far away from the root node.

Example:

In the below tree structure, we have shown the traversing of the tree using BFS algorithm from the root node S to goal node K. BFS search algorithm traverse in layers, so it will follow the path which is shown by the dotted arrow, and the traversed path will be:

1. S---> A--->B---->C--->D---->G--->H--->E---->F---->I---->K



Time Complexity: Time Complexity of BFS algorithm can be obtained by the number of nodes traversed in BFS until the shallowest Node. Where the d= depth of shallowest solution and b is a node at every state.

$T (b) = 1+b^2+b^3+.......+ b^d= O (b^d)$

Space Complexity: Space complexity of BFS algorithm is given by the Memory size of frontier which is $O(b^d)$.

Completeness: BFS is complete, which means if the shallowest goal node is at some finite depth, then BFS will find a solution.

Optimality: BFS is optimal if path cost is a non-decreasing function of the depth of the node.

2. Depth-first Search

- Depth-first search isa recursive algorithm for traversing a tree or graph data structure.
- It is called the depth-first search because it starts from the root node and follows each path to its greatest depth node before moving to the next path.
- DFS uses a stack data structure for its implementation.
- The process of the DFS algorithm is similar to the BFS algorithm.

Note: Backtracking is an algorithm technique for finding all possible solutions using recursion.

Advantage:

- DFS requires very less memory as it only needs to store a stack of the nodes on the path from root node to the current node.
- It takes less time to reach to the goal node than BFS algorithm (if it traverses in the right path).

Disadvantage:

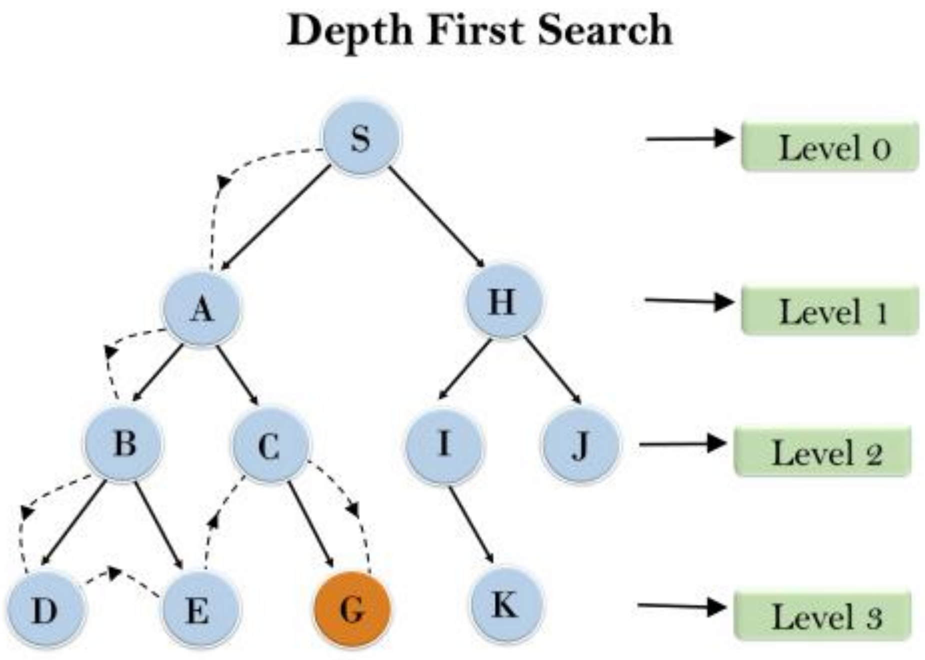
- There is the possibility that many states keep re-occurring, and there is no guarantee of finding the solution.
- DFS algorithm goes for deep down searching and sometime it may go to the infinite loop.

Example:

In the below search tree, we have shown the flow of depth-first search, and it will follow the order as:

Root node--->Left node ----> right node.

It will start searching from root node S, and traverse A, then B, then D and E, after traversing E, it will backtrack the tree as E has no other successor and still goal node is not found. After backtracking it will traverse node C and then G, and here it will terminate as it found goal node.



Completeness: DFS search algorithm is complete within finite state space as it will expand every node within a limited search tree.

Time Complexity: Time complexity of DFS will be equivalent to the node traversed by the algorithm. It is given by:

$T(n)= 1+ n^2+ n^3 +.........+ n^m=O(n^m)$

Where, m= maximum depth of any node and this can be much larger than d (Shallowest solution depth)

Space Complexity: DFS algorithm needs to store only single path from the root node, hence space complexity of DFS is equivalent to the size of the fringe set, which is **O(bm)**.

Optimal: DFS search algorithm is non-optimal, as it may generate a large number of steps or high cost to reach to the goal node.

3. Depth-Limited Search Algorithm:

A depth-limited search algorithm is similar to depth-first search with a predetermined limit. Depth-limited search can solve the drawback of the infinite path in the Depth-first search. In this algorithm, the node at the depth limit will treat as it has no successor nodes further.

Depth-limited search can be terminated with two Conditions of failure:

- Standard failure value: It indicates that problem does not have any solution.
- Cutoff failure value: It defines no solution for the problem within a given depth limit.

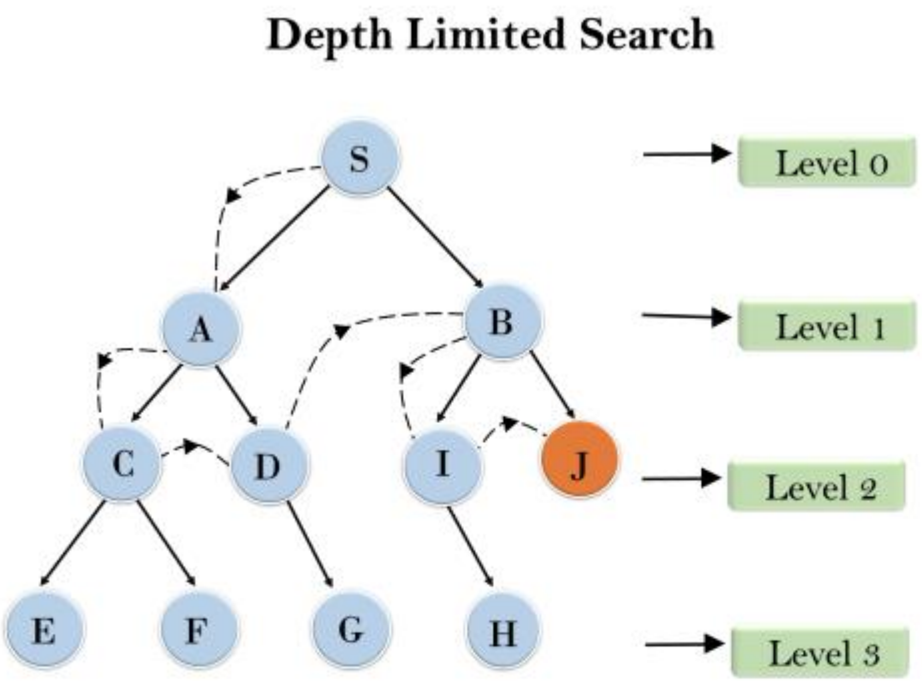
Advantages:

Depth-limited search is Memory efficient.

Disadvantages:

- Depth-limited search also has a disadvantage of incompleteness.
- It may not be optimal if the problem has more than one solution.

Example:



Completeness: DLS search algorithm is complete if the solution is above the depth-limit.

Time Complexity: Time complexity of DLS algorithm is **O(b^{ℓ})**.

Space Complexity: Space complexity of DLS algorithm is O**($b \times \ell$)**.

Optimal: Depth-limited search can be viewed as a special case of DFS, and it is also not optimal even if $\ell > d$.

4. Uniform-cost Search Algorithm:

Uniform-cost search is a searching algorithm used for traversing a weighted tree or graph. This algorithm comes into play when a different cost is available for each edge. The primary goal of the uniform-cost search is to find a path to the goal node which has the lowest cumulative cost. Uniform-cost search expands nodes according to their path costs form the root node. It can be used to solve any graph/tree where the optimal cost is in demand. A uniform-cost search algorithm is implemented by the priority queue. It gives maximum priority to the lowest cumulative cost. Uniform cost search is equivalent to BFS algorithm if the path cost of all edges is the same.

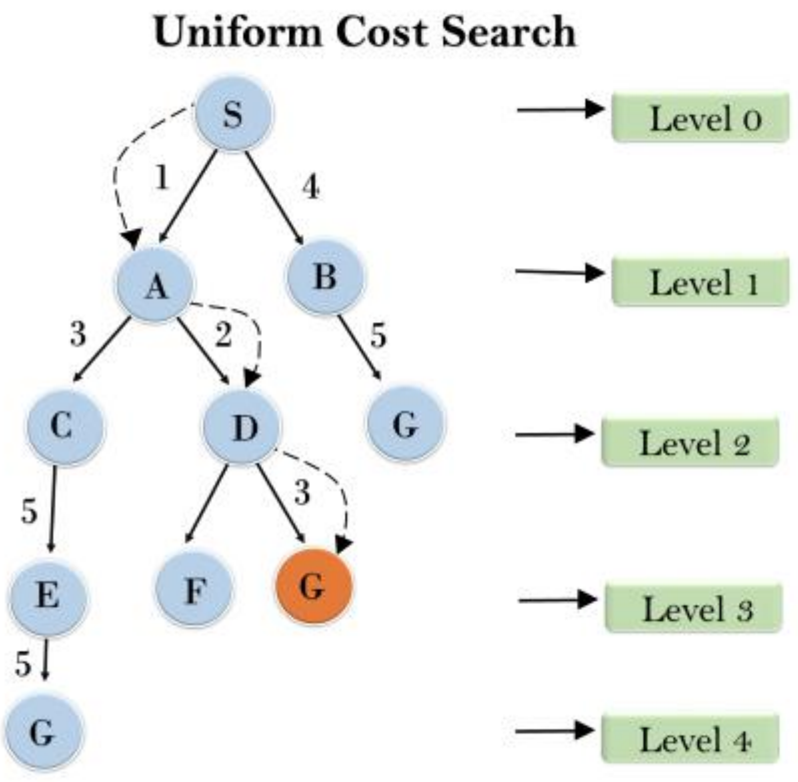
Advantages:

- Uniform cost search is optimal because at every state the path with the least cost is chosen.

Disadvantages:

- It does not care about the number of steps involve in searching and only concerned about path cost. Due to which this algorithm may be stuck in an infinite loop.

Example:



Completeness:

Uniform-cost search is complete, such as if there is a solution, UCS will find it.

Time Complexity:

Let C* **is Cost of the optimal solution**, and **ε** is each step to get closer to the goal node. Then the number of steps is = C*/ε+1. Here we have taken +1, as we start from state 0 and end to C*/ε.

Hence, the worst-case time complexity of Uniform-cost search is**$O(b^{1 + [C^*/\varepsilon]})$/**.

Space Complexity:

The same logic is for space complexity so, the worst-case space complexity of Uniform-cost search is **O($b^{1 + [C^*/ε]}$)**.

Optimal:

Uniform-cost search is always optimal as it only selects a path with the lowest path cost.

5. Iterative deepeningdepth-first Search:

The iterative deepening algorithm is a combination of DFS and BFS algorithms. This search algorithm finds out the best depth limit and does it by gradually increasing the limit until a goal is found.

This algorithm performs depth-first search up to a certain "depth limit", and it keeps increasing the depth limit after each iteration until the goal node is found.

This Search algorithm combines the benefits of Breadth-first search's fast search and depth-first search's memory efficiency.

The iterative search algorithm is useful uninformed search when search space is large, and depth of goal node is unknown.

Advantages:

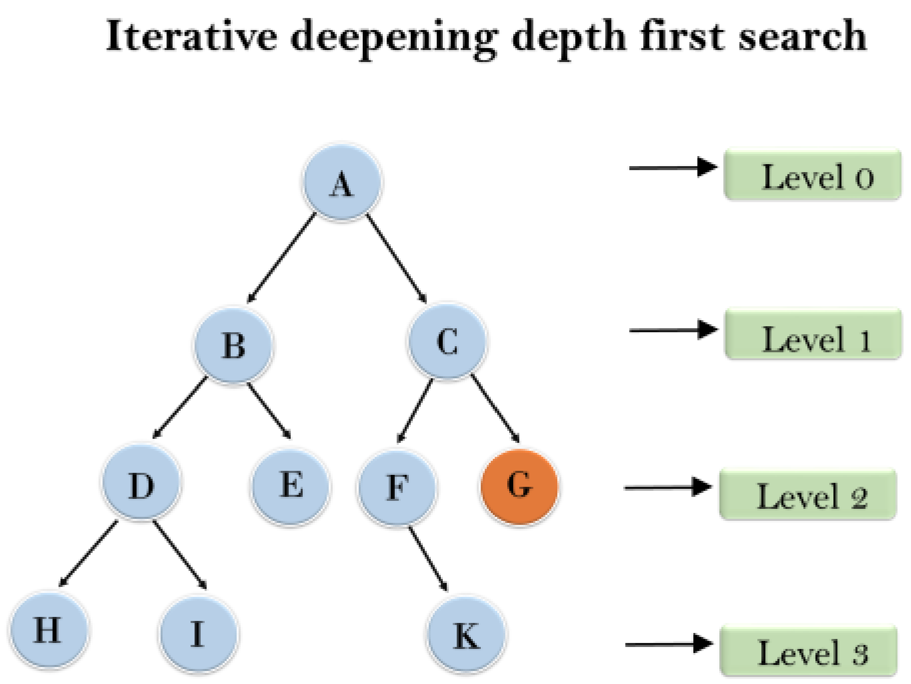
- It combines the benefits of BFS and DFS search algorithm in terms of fast search and memory efficiency.

Disadvantages:

- The main drawback of IDDFS is that it repeats all the work of the previous phase.

Example:

Following tree structure is showing the iterative deepening depth-first search. IDDFS algorithm performs various iterations until it does not find the goal node. The iteration performed by the algorithm is given as:



1'st Iteration-----> A
2'nd Iteration----> A, B, C
3'rd Iteration------>A, B, D, E, C, F, G
4'th Iteration------>A, B, D, H, I, E, C, F, K, G
In the fourth iteration, the algorithm will find the goal node.

Completeness:

This algorithm is complete is ifthe branching factor is finite.

Time Complexity:

Let's suppose b is the branching factor and depth is d then the worst-case time complexity is **O(b^d)**.

Space Complexity:

The space complexity of IDDFS will be **O(bd)**.

Optimal:

IDDFS algorithm is optimal if path cost is a non- decreasing function of the depth of the node.

6. Bidirectional Search Algorithm:

Bidirectional search algorithm runs two simultaneous searches, one form initial state called as forward-search and other from goal node called as backward-search, to find the goal node. Bidirectional search replaces one single search graph with two small

subgraphs in which one starts the search from an initial vertex and other starts from goal vertex. The search stops when these two graphs intersect each other.

Bidirectional search can use search techniques such as BFS, DFS, DLS, etc.

Advantages:

- Bidirectional search is fast.
- Bidirectional search requires less memory

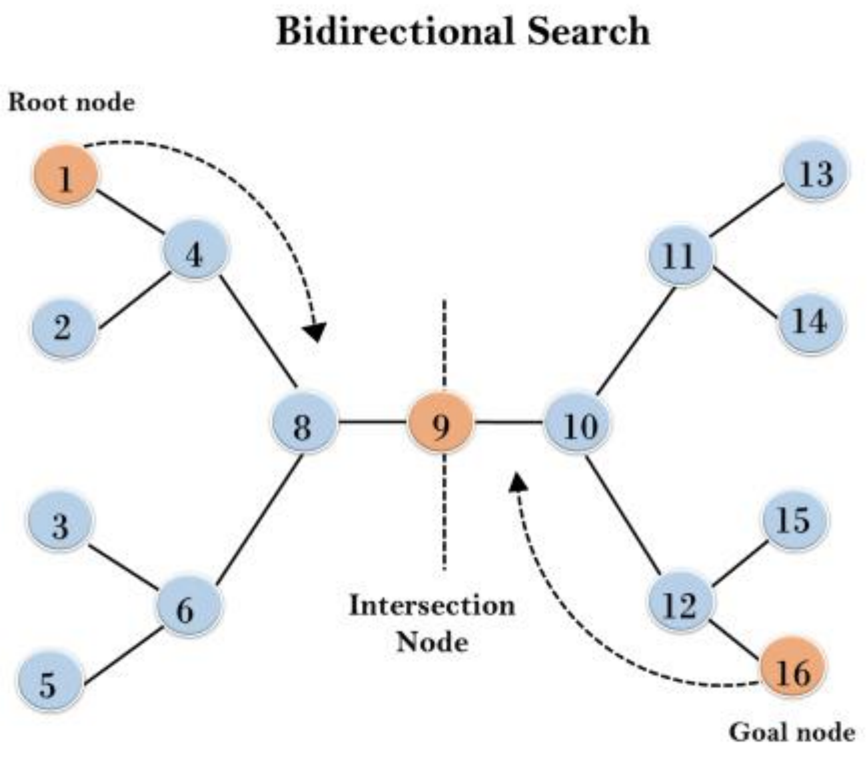
Disadvantages:

- Implementation of the bidirectional search tree is difficult.
- **In bidirectional search, one should know the goal state in advance.**

Example:

In the below search tree, bidirectional search algorithm is applied. This algorithm divides one graph/tree into two sub-graphs. It starts traversing from node 1 in the forward direction and starts from goal node 16 in the backward direction.

The algorithm terminates at node 9 where two searches meet.

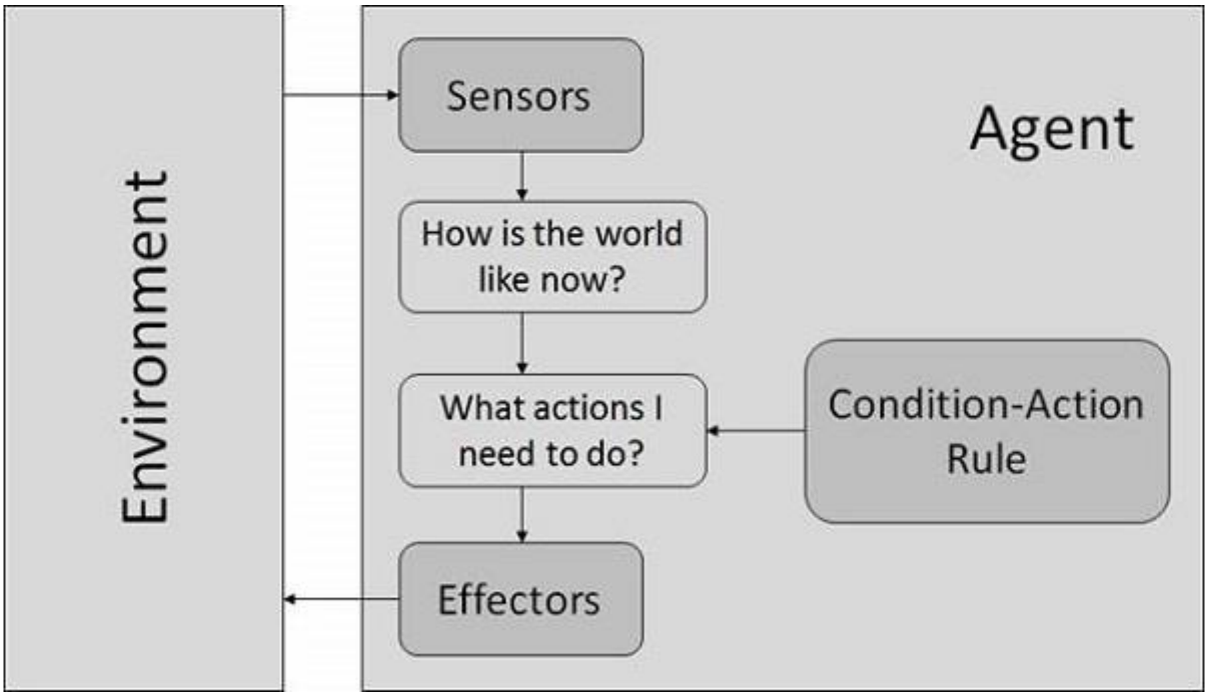


Completeness: Bidirectional Search is complete if we use BFS in both searches.

Time Complexity: Time complexity of bidirectional search using BFS is **$O(b^d)$**.

Space Complexity: Space complexity of bidirectional search is **O(b^d)**.

Optimal: Bidirectional search is Optimal.



Model Based Reflex Agents

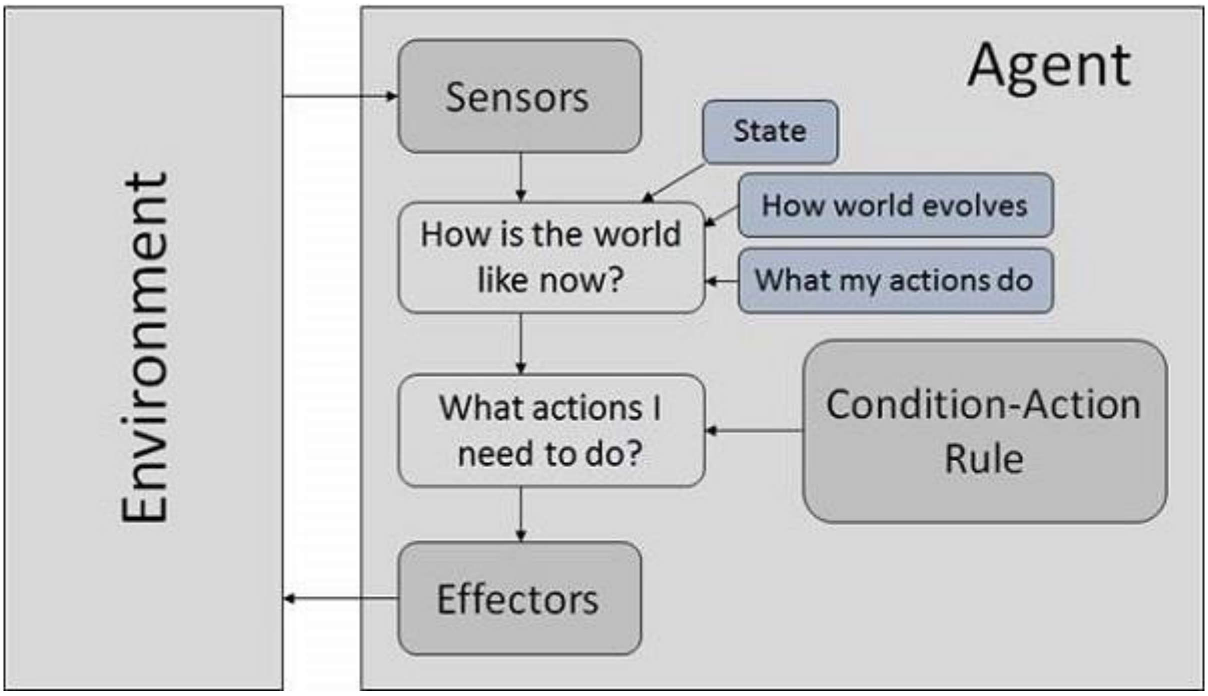
They use a model of the world to choose their actions. They maintain an internal state.

Model − knowledge about "how the things happen in the world".

Internal State − It is a representation of unobserved aspects of current state depending on percept history.

Updating the state requires the information about −

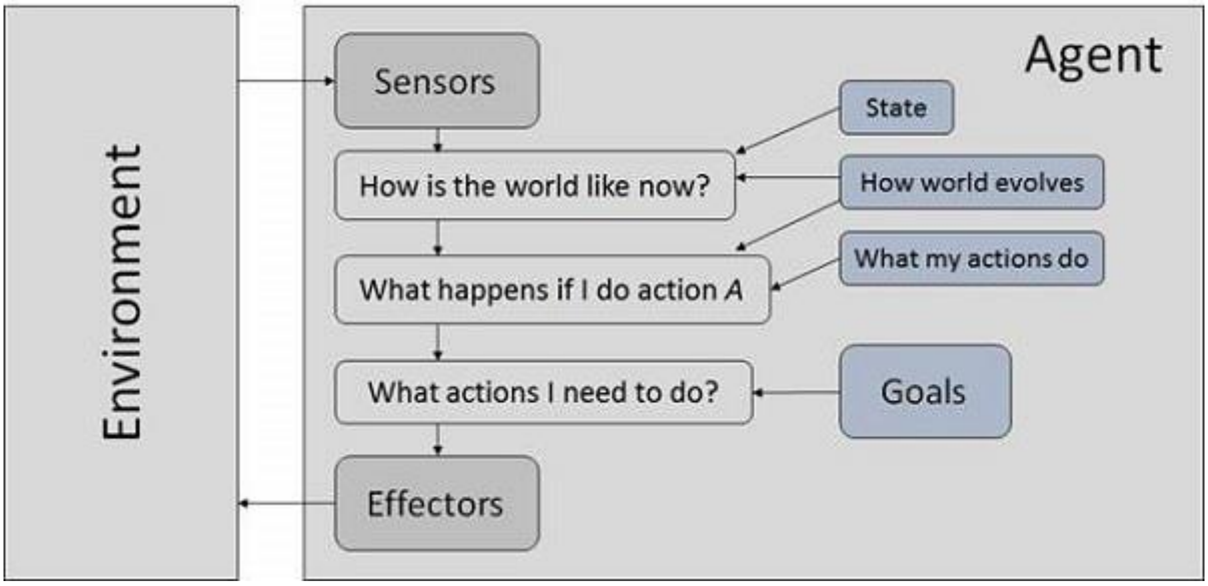
- How the world evolves.
- How the agent's actions affect the world.



Goal Based Agents

They choose their actions in order to achieve goals. Goal-based approach is more flexible than reflex agent since the knowledge supporting a decision is explicitly modeled, thereby allowing for modifications.

Goal − It is the description of desirable situations.


Agent
Environment
Sensors
State
How is the world like now?
How world evolves
What happens if I do action A
What my actions do
What actions I need to do?
Goals
Effectors

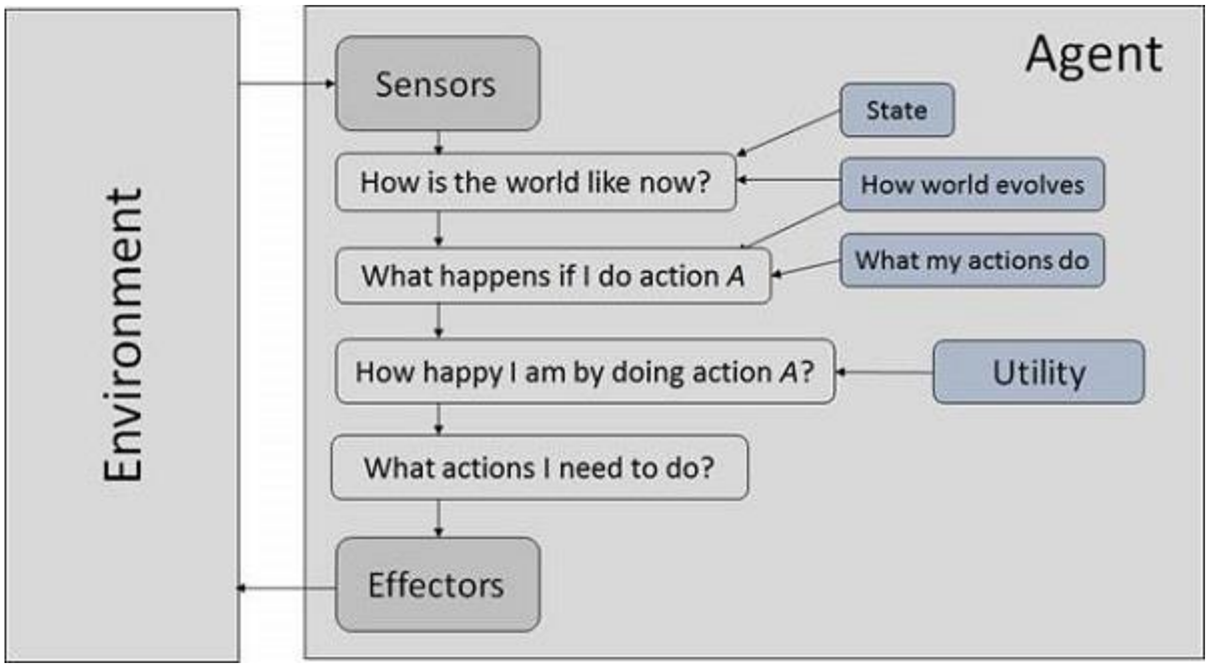
Utility Based Agents

They choose actions based on a preference (utility) for each state.

Goals are inadequate when −

- There are conflicting goals, out of which only few can be achieved.

- Goals have some uncertainty of being achieved and you need to weigh likelihood of success against the importance of a goal.


Environment
Agent
Sensors
State
How is the world like now?
How world evolves
What happens if I do action A
What my actions do
How happy I am by doing action A?
Utility
What actions I need to do?
Effectors

The Nature of Environments

Some programs operate in the entirely **artificial environment** confined to keyboard input, database, computer file systems and character output on a screen.

In contrast, some software agents (software robots or softbots) exist in rich, unlimited softbots domains. The simulator has a **very detailed, complex environment**. The software agent needs to choose from a long array of actions in real time. A softbot designed to scan the online preferences of the customer and show interesting items to the customer works in the **real** as well as an **artificial** environment.

The most famous **artificial environment** is the **Turing Test environment**, in which one real and other artificial agents are tested on equal ground. This is a very challenging environment as it is highly difficult for a software agent to perform as well as a human.

Turing Test

The success of an intelligent behavior of a system can be measured with Turing Test.

Two persons and a machine to be evaluated participate in the test. Out of the two persons, one plays the role of the tester. Each of them sits in different rooms. The tester is unaware of who is machine and who is a human. He interrogates the questions by typing and sending them to both intelligences, to which he receives typed responses.

This test aims at fooling the tester. If the tester fails to determine machine’s response from the human response, then the machine is said to be intelligent.

Properties of Environment

The environment has multifold properties −

- **Discrete / Continuous** − If there are a limited number of distinct, clearly defined, states of the environment, the environment is discrete (For example, chess); otherwise it is continuous (For example, driving).
- **Observable / Partially Observable** − If it is possible to determine the complete state of the environment at each time point from the percepts it is observable; otherwise it is only partially observable.
- **Static / Dynamic** − If the environment does not change while an agent is acting, then it is static; otherwise it is dynamic.
- **Single agent / Multiple agents** − The environment may contain other agents which may be of the same or different kind as that of the agent.
- **Accessible / Inaccessible** − If the agent's sensory apparatus can have access to the complete state of the environment, then the environment is accessible to that agent.
- **Deterministic / Non-deterministic** − If the next state of the environment is completely determined by the current state and the actions of the agent, then the environment is deterministic; otherwise it is non-deterministic.
- **Episodic / Non-episodic** − In an episodic environment, each episode consists of the agent perceiving and then acting. The quality of its action depends just on the episode itself. Subsequent episodes do not depend on the actions in the previous episodes. Episodic environments are much simpler because the agent does not need to think ahead.

Heuristic Search

All of the search methods in the preceding section are uninformed in that they did not take into account the goal. They do not use any information about where they are trying to get to unless they happen to stumble on a goal. One form of heuristic information about which nodes seem the most promising is a heuristic function *h(n)*, which takes a node *n* and returns a non-negative real number that is an estimate of the path cost from node *n* to a goal node. The function *h(n)* is an *underestimate* if *h(n)* is less than or equal to the actual cost of a lowest-cost path from node *n* to a goal.
The heuristic function is a way to inform the search about the direction to a goal. It provides an informed way to guess which neighbor of a node will lead to a goal.
There is nothing magical about a heuristic function. It must use only information that can be readily obtained about a node. Typically a trade-off exists between the amount of work it takes to derive a heuristic value for a node and how accurately the heuristic value of a node measures the actual path cost from the node to a goal.
A standard way to derive a heuristic function is to solve a simpler problem and to use the actual cost in the simplified problem as the heuristic function of the original problem.
Example 3.12: For the graph of Figure 3.2, the straight-line distance in the world between the node and the goal position can be used as the heuristic function. The examples that follow assume the following heuristic function:

h(mail)	=	26	h(ts)	=	23	h(o103)	=	21
h(o109)	=	24	h(o111)	=	27	h(o119)	=	11
h(o123)	=	4	h(o125)	=	6	h(r123)	=	0
h(b1)	=	13	h(b2)	=	15	h(b3)	=	17

h(b4)	=	18	h(c1)	=	6	h(c2)	=	10
h(c3)	=	12	h(storage)	=	12			

This *h* function is an underestimate because the *h* value is less than or equal to the exact cost of a lowest-cost path from the node to a goal. It is the exact cost for node *o123*. It is very much an underestimate for node *b1*, which seems to be close, but there is only a long route to the goal. It is very misleading for *c1*, which also seems close to the goal, but no path exists from that node to the goal.

Example 3.13: Consider the delivery robot of Example 3.2, where the state space includes the parcels to be delivered. Suppose the cost function is the total distance traveled by the robot to deliver all of the parcels. One possible heuristic function is the largest distance of a parcel from its destination. If the robot could only carry one parcel, a possible heuristic function is the sum of the distances that the parcels must be carried. If the robot could carry multiple parcels at once, this may not be an underestimate of the actual cost.

The *h* function can be extended to be applicable to (non-empty) paths. The heuristic value of a path is the heuristic value of the node at the end of the path. That is:

$h(\langle n_o,...,n_k \rangle)=h(n_k)$

A simple use of a heuristic function is to order the neighbors that are added to the stack representing the frontier in depth-first search. The neighbors can be added to the frontier so that the best neighbor is selected first. This is known as **heuristic depth-first search**. This search chooses the locally best path, but it explores all paths from the selected path before it selects another path. Although it is often used, it suffers from the problems of depth-fist search.
Another way to use a heuristic function is to always select a path on the frontier with the lowest heuristic value. This is called **best-first search**. It usually does not work very well; it can follow paths that look promising because they are close to the goal, but the costs of the paths may keep increasing.

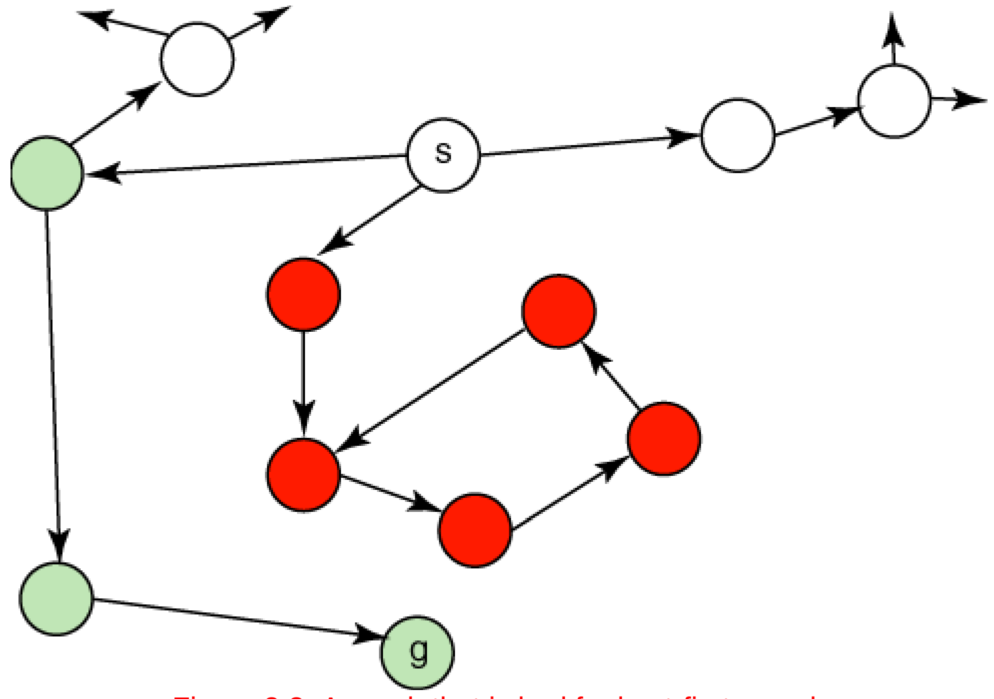


Figure 3.8: A graph that is bad for best-first search

Example 3.14: Consider the graph shown in Figure 3.8, where the cost of an arc is its length. The aim is to find the shortest path from *s* to *g*. Suppose the Euclidean distance to the goal *g* is used as the heuristic function. A heuristic depth-first search will select the node below *s* and will never terminate. Similarly, because all of the nodes below *s* look good, a best-first search will cycle between them, never trying an alternate route from *s*.

Logistic Regression

As I said earlier, fundamentally, Logistic Regression is used to classify elements of a set into two groups (binary classification) by calculating the probability of each element of the set.

Steps of Logistic Regression

In logistic regression, we decide a probability threshold. If the probability of a particular element is higher than the probability threshold then we classify that element in one group or vice versa.

Step 1

To calculate the binary separation, first, we determine the best-fitted line by following the Linear Regression steps.

Step 2

The regression line we get from Linear Regression is highly susceptible to outliers. Thus it will not do a good job in classifying two classes.

Thus, the predicted value gets converted into probability by feeding it to the sigmoid function.

The equation of sigmoid:

As we can see in Fig 3, we can feed any real number to the sigmoid function and it will return a value between 0 and 1.

Fig 2: Sigmoid curve (picture taken from Wikipedia)

Thus, if we feed the output **ŷ** value to the sigmoid function it retunes a probability value between 0 and 1.

Step 3

Finally, the output value of the sigmoid function gets converted into 0 or 1(discreet values) based on the threshold value. We usually set the threshold value as 0.5. In this way, we get the binary classification.

Now as we have the basic idea that how Linear Regression and Logistic Regression are related, let us revisit the process with an example.

Example
Let us consider a problem where we are given a dataset containing Height and Weight for a group of people. Our task is to predict the Weight for new entries in the Height column.

So we can figure out that this is a regression problem where we will build a Linear Regression model. We will train the model with provided Height and Weight values. Once the model is trained we can predict Weight for a given unknown Height value.

Fig 3: Linear Regression

Now suppose we have an additional field **Obesity** and we have to classify whether a person is obese or not depending on their provided height and weight. This is clearly a classification problem where we have to segregate the dataset into two classes (Obese and Not-Obese).

So, for the new problem, we can again follow the Linear Regression steps and build a regression line. This time, the line will be based on two parameters Height and Weight and the regression line will fit between two discreet sets of values. As this regression line is highly susceptible to outliers, it will not do a good job in classifying two classes.

To get a better classification, we will feed the output values from the regression line to the sigmoid function. The sigmoid function returns the probability for each output value from the regression line. Now based on a predefined threshold value, we can easily classify the output into two classes Obese or Not-Obese.

Fig 4: Linear Regression Vs Logistic Regression
Finally, we can summarize the similarities and differences between these two models.

The Similarities between Linear Regression and Logistic Regression

- Linear Regression and Logistic Regression both are supervised Machine Learning algorithms.
- Linear Regression and Logistic Regression, both the models are parametric regression i.e. both the models use linear equations for predictions

That’s all the similarities we have between these two models.

However, functionality-wise these two are completely different. Following are the differences.

- Linear Regression is used to handle regression problems whereas Logistic regression is used to handle the classification problems.

- Linear regression provides a continuous output but Logistic regression provides discreet output.
- The purpose of Linear Regression is to find the best-fitted line while Logistic regression is one step ahead and fitting the line values to the sigmoid curve.

Gradient Descent

Let’s say you are playing a game where the players are at the top of a mountain, and they are asked to reach the lowest point of the mountain. Additionally, they are blindfolded. So, what approach do you think would make you reach the lake?

Take a moment to think about this before you read on.

The best way is to observe the ground and find where the land descends. From that position, take a step in the descending direction and iterate this process until we reach the lowest point.

Finding the lowest point in a hilly landscape. (Source: Fisseha Berhane)

Gradient descent is an iterative optimization algorithm for finding the local minimum of a function.

To find the local minimum of a function using gradient descent, we must take steps proportional to the negative of the gradient (move away from the gradient) of the function at the current point. If we take steps proportional to the positive of the gradient (moving towards the gradient), we will approach a local maximum of the function, and the procedure is called **Gradient Ascent.**

Gradient descent was originally proposed by **CAUCHY** in 1847. It is also known as steepest descent.

Source: Clairvoyant

The goal of the gradient descent algorithm is to minimize the given function (say cost function). To achieve this goal, it performs two steps iteratively:

1. **Compute the gradient** (slope), the first order derivative of the function at that point
2. **Make a step (move) in the direction opposite to the gradient**, opposite direction of slope increase from the current point by alpha times the gradient at that point

Source: Coursera

Alpha is called **Learning rate –** a tuning parameter in the optimization process. It decides the length of the steps.

Plotting the Gradient Descent Algorithm

When we have a single parameter (theta), we can plot the dependent variable cost on the y-axis and theta on the x-axis. If there are two parameters, we can go with a 3-D plot, with cost on one axis and the two parameters (thetas) along the other two axes.

cost along z-axis and parameters(thetas) along x-axis and y-axis (source: Research gate)

It can also be visualized by using **Contours.** This shows a 3-D plot in two dimensions with parameters along both axes and the response as a contour. The value of the response increases away from the center and has the same value along with the rings. The response is directly proportional to the distance of a point from the center (along a direction).

Gradient descent using Contour Plot. (source: Coursera)

Alpha – The Learning Rate

We have the direction we want to move in, now we must decide the size of the step we must take.

***It must be chosen carefully to end up with local minima.**

- If the learning rate is too high, we might **OVERSHOOT** the minima and keep bouncing, without reaching the minima
- If the learning rate is too small, the training might turn out to be too long

Source: Coursera

1. a) Learning rate is optimal, model converges to the minimum
2. b) Learning rate is too small, it takes more time but converges to the minimum
3. c) Learning rate is higher than the optimal value, it overshoots but converges ($1/C < \eta < 2/C$)
4. d) Learning rate is very large, it overshoots and diverges, moves away from the minima, performance decreases on learning

Source: researchgate

<u>Note:</u> *As the gradient decreases while moving towards the local minima, the size of the step decreases. So, the learning rate (alpha) can be constant over the optimization and need not be varied iteratively.*

Local Minima

The cost function may consist of many minimum points. The gradient may settle on any one of the minima, which depends on the initial point (i.e initial parameters(theta)) and the learning rate. Therefore, the optimization may converge to different points with different starting points and learning rate.

Convergence of cost function with different starting points (Source: Gfycat)

-

Code Implementation of Gradient Descent in Python

Gradient Descent Algorithm

-

End Notes

Once we tune the learning parameter (alpha) and get the optimal learning rate, we start iterating until we converge to the local minima.

- stimation.

K-means clustering

K-means clustering is one of the simplest and popular unsupervised machine learning algorithms.

Typically, unsupervised algorithms make inferences from datasets using only input vectors without referring to known, or labelled, outcomes.

AndreyBu, who has more than 5 years of machine learning experience and currently teaches people his skills, says that "the objective of K-means is simple: group similar data points together and discover underlying patterns. To achieve this objective, K-means looks for a fixed number (*k*) of clusters in a dataset."

A cluster refers to a collection of data points aggregated together because of certain similarities.

You'll define a target number *k*, which refers to the number of centroids you need in the dataset. A centroid is the imaginary or real location representing the center of the cluster.

Every data point is allocated to each of the clusters through reducing the in-cluster sum of squares.

In other words, the K-means algorithm identifies *k* number of centroids, and then allocates every data point to the nearest cluster, while keeping the centroids as small as possible.

The *'means'* in the K-means refers to averaging of the data; that is, finding the centroid.

How the K-means algorithm works

To process the learning data, the K-means algorithm in data mining starts with a first group of randomly selected centroids, which are used as the beginning points for every cluster, and then performs iterative (repetitive) calculations to optimize the positions of the centroids

It halts creating and optimizing clusters when either:

- The centroids have stabilized — there is no change in their values because the clustering has been successful.
- The defined number of iterations has been achieved.

K-means algorithm example problem

Let's see the steps on how the K-means machine learning algorithm works using the Python programming language.

We'll use the Scikit-learn library and some random data to illustrate a K-means clustering simple explanation.

Step 1: Import libraries

```
import pandas as pdimport numpy as npimport matplotlib.pyplot as pltfrom sklearn.cluster
import KMeans%matplotlib inline
```

As you can see from the above code, we'll import the following libraries in our project:

- Pandas for reading and writing spreadsheets
- Numpy for carrying out efficient computations
- Matplotlib for visualization of data

Step 2: Generate random data

Here is the code for generating some random data in a two-dimensional space:

```
X= -2 * np.random.rand(100,2)X1 = 1 + 2 * np.random.rand(50,2)X[50:100, :] = 
X1plt.scatter(X[ : , 0], X[ :, 1], s = 50, c = 'b')plt.show()
```

A total of 100 data points has been generated and divided into two groups, of 50 points each.

Here is how the data is displayed on a two-dimensional space:

Step 3: Use Scikit-Learn

We'll use some of the available functions in the Scikit-learn library to process the randomly generated data.

Here is the code:

```
from sklearn.cluster import KMeansKmean = KMeans(n_clusters=2)Kmean.fit(X)
```

In this case, we arbitrarily gave *k* (n_clusters) an arbitrary value of two.

Here is the output of the K-means parameters we get if we run the code:

```
KMeans(algorithm='auto', copy_x=True, init='k-means++', max_iter=300

 n_clusters=2, n_init=10, n_jobs=1, precompute_distances='auto',

 random_state=None, tol=0.0001, verbose=0)
```

Step 4: Finding the centroid

Here is the code for finding the center of the clusters:

```
Kmean.cluster_centers_
```

Here is the result of the value of the centroids:

```
array([[-0.94665068, -0.97138368],

 [ 2.01559419, 2.02597093]])
```

Let's display the cluster centroids (using green and red color).

```
plt.scatter(X[ : , 0], X[ : , 1], s =50, c='b')plt.scatter(-0.94665068, -0.97138368, s=200, c='g',
marker='s')plt.scatter(2.01559419, 2.02597093, s=200, c='r', marker='s')plt.show()
```

Here is the output:

Step 5: Testing the algorithm

Here is the code for getting the labels property of the K-means clustering example dataset; that is, how the data points are categorized into the two clusters.

```
Kmean.labels_
```

Here is the result of running the above K-means algorithm code:

```
array([0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,

 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0,
 0, 0, 0, 0, 0, 0, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1,

 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1, 1])
```

As you can see above, 50 data points belong to the **0** cluster while the rest belong to the **1** cluster.

For example, let's use the code below for predicting the cluster of a data point:

```
sample_test=np.array([-3.0,-3.0])second_test=sample_test.reshape(1, -1)Kmean.predict(second_test)
```

Here is the result:

```
array([0])
```

It shows that the test data point belongs to the **0** (green centroid) cluster.

Wrapping up

Here is the entire K-means clustering algorithm code in Python:

```
import pandas as pdimport numpy as npimport matplotlib.pyplot as pltfrom sklearn.cluster import KMeans%matplotlib inlineX= -2 * np.random.rand(100,2)X1 = 1 + 2 * np.random.rand(50,2)X[50:100, :] = X1plt.scatter(X[ : , 0], X[ :, 1], s = 50, c = 'b')plt.show()from sklearn.cluster import KMeansKmean = KMeans(n_clusters=2)Kmean.fit(X)Kmean.cluster_centers_plt.scatter(X[ : , 0], X[ : , 1], s =50, c='b')plt.scatter(-0.94665068, -0.97138368, s=200, c='g', marker='s')plt.scatter(2.01559419, 2.02597093, s=200, c='r', marker='s')plt.show()Kmean.labels_sample_test=np.array([-3.0,-3.0])second_test=sample_test.reshape(1, -1)Kmean.predict(second_test)
```

K-means clustering is an extensively used technique for data cluster analysis.

It is easy to understand, especially if you accelerate your learning using a K-means clustering tutorial. Furthermore, it delivers training results quickly.

However, its performance is usually not as competitive as those of the other sophisticated clustering techniques because slight variations in the data could lead to high variance.

Furthermore, clusters are assumed to be spherical and evenly sized, something which may reduce the accuracy of the K-means clustering Python results.

References

1. Stuart Russel, Peter Norvig, “Artificial Intelligence – A Modern Approach”, Second Edition, PHI/Pearson Education.
2. Patrick Henry Winston, “Artificial Intelligence”, 3rd Edition, Pearson Education.
3. Kevin P. Murphy. “Machine Learning: A Probabilistic Perspective”.
MIT Press, 2012. Christopher Bishop. “Pattern Recognition and Machine Learning”, First Edition,Springer, 2006.
4. R. Duda, P. Hart and D. Stork. “Pattern Classification” Second Edition, John Wiley & Sons INC., 2001.
5. Tom Mitchell. “Machine Learning” First Edition, McGraw-Hill, 1997.