Artificial Intelligence – Making an Intelligent personal assistant

Mr. Ankush Bhatia

U & P. U Patel Department of Computer Engineering, Charotar University of Science and Technology, Anand, Gujarat, 388421, India.
13ce010@charusat.edu.in

Abstract

A bot in computing is an autonomous program on a network (especially the Internet) which can interact with systems or users.[Simpson, J., and Weiner, E. (1989)**]** This document gives the description of how memory of an Artificial-Intelligence bot can be stored in an optimized way with a faster searching algorithm and how it can learn new things; the user wants the bot to learn. This paper gives the details of how a bot uses a an ordered tree data structure, called TRIE or a prefix tree to dynamically store the things it learns and what to reply when a person commands asks him something, with a little modification.

Keywords— Bot, Python, Prefix Tree, Trie, Web Crawler, Web Indexing, Web Scraper, Web Spider**.**

1. INTRODUCTION

ARTIFICIAL INTELLIGENCE (AI) is a field in computing which deals with the study/creating computer systems or software which show intelligent behavior. In general terms, Artificial Intelligence is the intelligent behavior exhibited by software or a system.[Goebel, Randy and Mackworth, Alan and Poole, David (1998)] An AI bot is an autonomous program on a network which can interact with systems or users, especially one designed to behave like a player in some computer games. The most common example of an AI bot is when you play Tic-Tac-Toe with a computer. The computer there is a bot which is made to play Tic-Tac-Toe. A bot is programmed in such a way, that it is able to decide where to put an X/O in the grid to win the game.

Human-Level Artificial Intelligence is the intelligence of a (hypothetical) machine that could successfully perform any intellectual task that a human being can.[Kurzweil, Ray (2005)] There have been many studies on the approach to achieve human-level intelligence in systems; one of them is called the Whole brain emulation (WBE). A WBE or mind uploading is the hypothetical process of copying mental content from a particular brain substrate and copying it to a computational device, such as a digital, analog, quantum-based or software-based artificial neural network.[Bamford, Simeon (2012)] But it is still a theory and a possible approach to achieve Human-Level AI in a personal assistant. One of the approaches to make a bot self-evolving and a constantly learning one are discussed here in the paper.

2. MEMORY OF A BOT

In the development of a bot, designing its memory is a crucial process. Its memory should be designed be in such a way that it takes less space and the process to retrieve data from its memory should be as quick as possible.

Designing the memory of a bot is a difficult task when the bot needs to be more humanly. For that, let's think humanly. What happens in our mind (human mind) when someone says, let's say for example, "Light". We start thinking about everything that is related to the word "Light" i.e. a huge portal containing everything related to "Light" opens up in our mind.

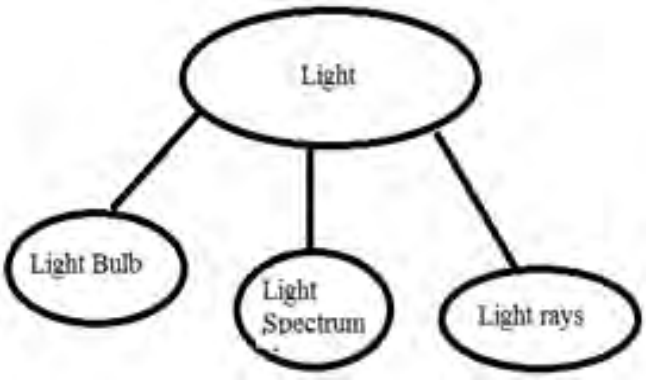

Fig. 1. Nodes related to "Light"

And then if he says “bulb” we start making a figure of a “Light Bulb” in our mind and everything that is related to a “Light Bulb” (See Fig. 2). A bot’s memory should do something like that too.

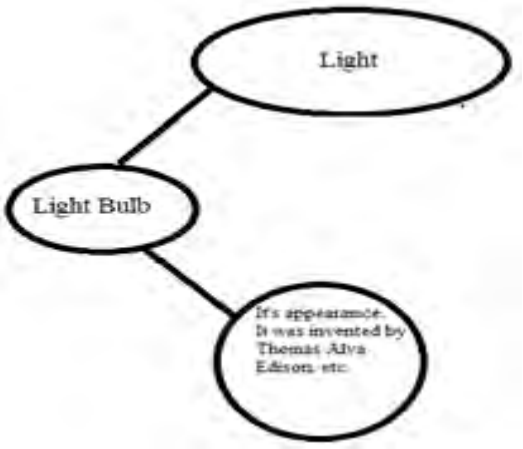

Fig. 2. How a human-mind reacts when it listens “Light Bulb”.

So the “Light Bulb” should be stored in a bot’s memory exactly like it is stored inside a human’s mind. For such mapping, in computer science, there is an abstract data type called Tree. In computer science, a graph is an abstract data type which is used to store a set of nodes (or vertices) where some pairs of nodes are connected by links (or edges) and a graph without the cycles is called a tree. In the above figure, “Light” is a node and the link between “Light” and “Light Bulb” is called an edge. But such mapping for every word can make a tree too complex and it’ll require a lot of memory. For that problem let us consider two simple words or nodes, “Light” and “Light Bulb”. In a graph these two nodes would look like:

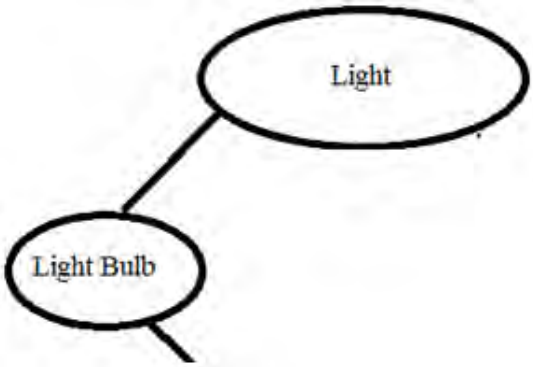

Fig. 3. “Light” and “Light Bulb” nodes in a graph.

Here “Light” and “Light Bulb” are two nodes in a tree or the memory of the bot, considering the size of the two words the size of the two nodes would be 5 + 9 = 14 (5 for “Light” and 9 for “Light Bulb”) bytes. But the word “Light” and “Light Bulb” share the common prefix “Light”, so instead of storing “Light” again in “Light Bulb”, “Light” should be used once which would lessen the size of the memory of the tree. To store the strings (words) with common prefixes an ordered tree data structure called Trie or Prefix tree is used. In computer science, a Trie, also called digital tree and sometimes radix tree or prefix tree (as they can be searched by prefixes), is an ordered tree data structure that is used to store a dynamic set or associative array where the keys are usually strings.[Black, Paul E. (2009)] A Trie to store “Light Bulb “ and “Light” would look like:

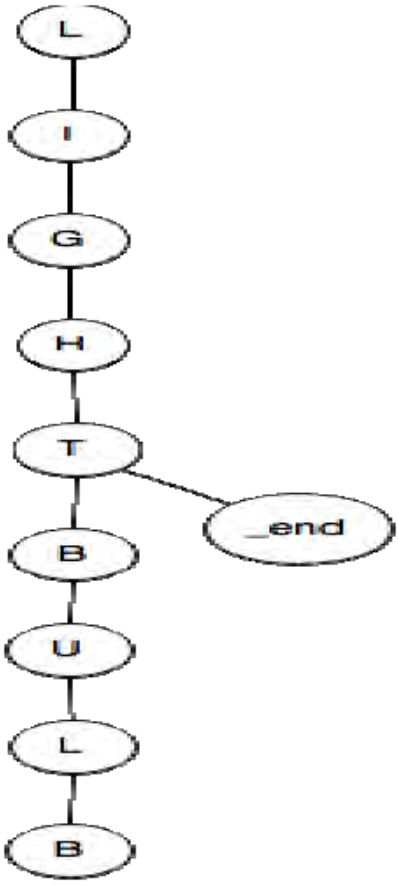

Fig. 4. "Light" and "Light Bulb" nodes in a Trie.

The "_end" node in the above figure describes that all the words above the "_end" node is a valid string. The size of the tree to store "Light Bulb" and "Light" would be 9, which is quite less as compared to a normal tree. And the time complexity for searching the string is O(n) where n is the length of the string to be searched. The implementation of the above tree in Python 2.7 is shown in Fig. 5.

```
#Memory of bot
class memory:
    _end = '__end'
    root = dict()
    def __init__(self, word):
        current_node = self.root
        for letter in word:
            current_node = current_node.setdefault(letter, {})
        current_node[self._end] = self._end
    def insert(self, word):
        current_node = self.root
        for letter in word:
            if letter in current_node:
                current_node = current_node[letter]
            else:
                current_node = current_node.setdefault(letter, {})
        current_node = current_node.setdefault(self._end, self._end)
```

Fig. 5. Python implementation of Trie.

But this was only the storing part, a bot needs to response to the command it has been given. For that we have to modify our Trie a bit more. In general, a Trie is used for two operations *viz.* insert and query. But for the query operation in our case we need some extra nodes which will result in the spontaneous response of the bot. So for the query operation I added two new nodes *viz.* "do" and "speak".

2.1 The "Do" node

Let's suppose the user gave the command to the bot to "open camera" and the bot should turn on the camera instantly. For that purpose I added an extra node along with every valid string inside the bot's memory or the Trie called the "do" node. The "do" node contains the name of function which should be called once that string is being queried by the user. The "do" quite rightly shown in the Fig. 6.

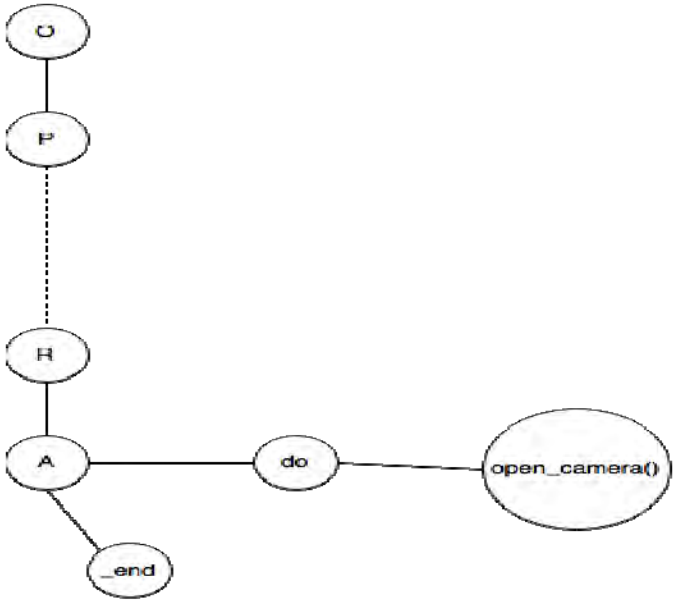

Fig. 6. The “do” node for command “open camera”

The Fig. 6 shows how the “do” node is added in the memory after every valid string or command (“_end” node depicts the end of a string). So when the user gives the command “open camera” to bot, the bot searches the string “open camera” in the memory. When the string is found the bot gets the details out of the “do” node and calls the function “open_camera()” which will open the camera. But there are some commands in which a bot is supposed to reply by saying something and not perform any action. For such commands I added a new node called “speak” node.

2.2 The “speak” node

Just like the “do” node, the “speak” node contains the string which a bot should reply when that string or command is queried by the user. For instance, let’s take a simple string “Hello”. When the user says “Hello”, the bot should reply “Hello, how are you?”. Fig. 7 shows how the “speak” node is added in the Trie or the bot’s memory.

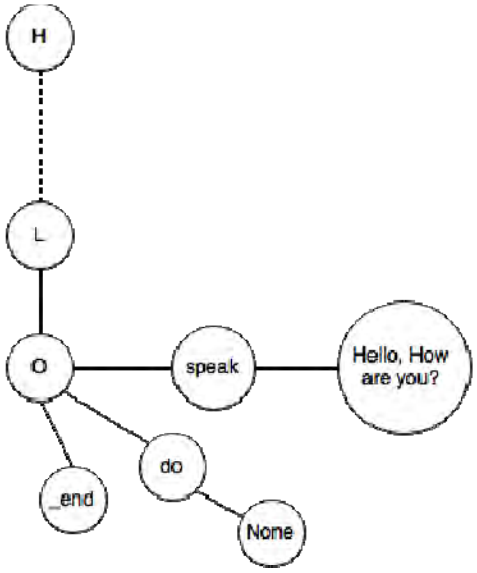

Fig. 7. The “speak” node for command “Hello”

In the Fig. 7 the “do” node contains a “None” value that means it doesn’t have to perform any actions. In some commands there might be a value in the “do” node and “speak” node. The implementation of the “do” and “speak” node and string search in a Trie in Python is shown in Fig. 8 on page number 4.

```
def search(self, word):                          #search in Trie
    current_node = self.root                     #root is the original Trie
    for letter in word:                          #word is the string to be searched
        if letter in current_node:
            current_node = current_node[letter]
        else:
            return False
    else:
        if self._end in current_node:
            return current_node['do'], current_node['speak'] #Tuple that returns "do"
        else:                                                #and "speak" value
            return None
def assign_do(self, word, function):            #Function to assign value to "do" node
    current_node = self.root                    #"function" contains name of function
    for letter in word:
        if letter in current_node:
            current_node = current_node[letter]
    current_node['do'] = function
def assign_speak(self, word, string):           #Function to assign value to "speak" node
    current_node = self.root                    #"string" contains string
    for letter in word:
        if letter in current_node:
            current_node = current_node[letter]
    current node['speak'] = string
```

Fig. 8. Python Implementation to search and assign value to “do” and “speak” nodes. The above piece of program is in continuation with program in Fig. 5.

The above program in Python 2.7 gives the implementation of the “do” and the “speak” node in the Trie. This was just the memory of the bot, but the important question is how to make it a self-learning bot. The above topic is discussed below.

3. MAKING A BOT, SELF-EVOLVING ONE

A good personal-assistant or the “bot” should be a self-evolving one and it should become more of what its user demands to. A good personal-assistant should be just like its user. An approach to make the bot a self-evolving one is discussed in this topic.

To make a bot self-evolving one, let’s think more humanly. When a human is born, he/she doesn’t know anything and as he keeps getting older he/she starts learning new things which get stored in his mind or memory. In the above topic we discussed how everything is stored in the memory of the bot such that its memory is as humanly as possible. But now, the question arises that how to make it a self-evolving bot. As humans keep learning from their sources of information like books, other humans, internet, etc. Since a bot is a virtual one, it can’t read or see things so the only source left for the bot is the internet. So for the bot to get information from the internet, it needs to crawl on the websites on the internet and scrap out the important information. For such tasks, a web-crawler or a spider for crawling and a web-scraper for scraping are used.

3.1 Web-Crawler

A Web-Crawler is itself a bot which browses over some websites or the entire internet. Web-crawler is also known as a Web-spider. For a personal assistant, to crawl over the entire internet is a bit overload of work, a bot can crawl over few sites and get the information.[Kobayashi, M. and Takeda, K. (2000)]

Let’s understand what a web crawler is. Let’s suppose we have a list of websites. Starting from the first website a web-spider goes through all the links in the webpages as in an actual user goes through until it gets the information. Look at the following flow chart (Fig. 9).

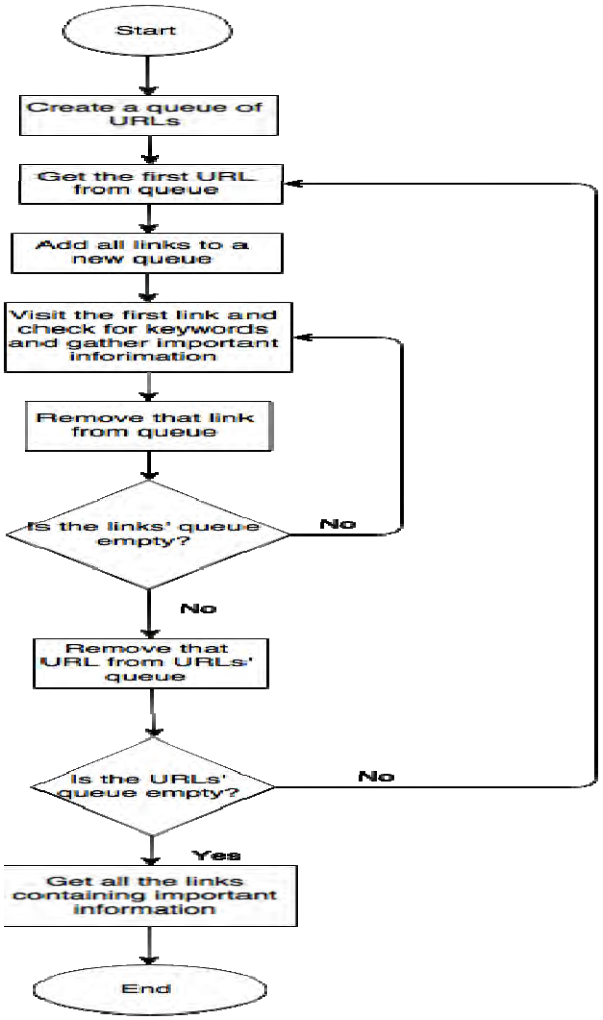

Fig. 9. Flow chart to explain the working of a simple web-crawler.

Now, let's understand what's happening with the help of flow chart in Fig. 9. We have a queue of all the important URLs (URLs of only home page). We'll start off with the first URL and extract all the links from the homepage by checking all the "href" tags in the html source of the page. Once we have all the links of the page we'll start visiting each link and check for the information that user has asked the bot. If the spider gets the results it'll store the link in a document and keep checking all the links until all the links are visited. This process will continue for each URL from the queue.

So, for that, I chose only a queue of 15-20 sites unlike any other search engine which crawl over the entire internet. The other important thing is prioritization of each URL. A URL with the most information should be first and the URL with the least information should be last. This way, the bot will get result much faster. Here is a Python implementation of a simple web crawler; it only lists out the links in a file.

```
import re
import urllib
links = open('links.txt','w')          #Text file to write the links
urlqueue = ["http://www.wikipedia.org/", "http://www.python.org/",
            "http://www.processing.org/"]   #URLs queue
for url in urlqueue:
    for i in re.findall('''href=["'](.[^"']+)["']''', urllib.urlopen(url).read(), re.I):
        print i
        for ee in re.findall('''href=["'](.[^"']+)["']''', urllib.urlopen(i).read(), re.I):
            print ee
            links.write(ee+'\n')        #Write each link in links.txt
links.close()
```

Fig. 10. Python Implementation of a naïve web-crawler.

3.2 Web-Scraper

A web-crawler can only get the links where the information is stored, to scrape out the information from that webpage; we need another bot, called the web-scraper. A web-scraper is a bot that scrapes out the data from a web page (html page). Since a web-crawler can only get the links which contains the data, the web-scraper scrapes out the data from those links and stores it into the memory. So, if something new is asked by the user the bot checks it on the internet and scrapes out the answer from the internet and stores it in memory so that, if the user asks again the same question the bot doesn't have to looking for the answers over the internet, the response will be instant. To understand how a web-scraper scrapes out the important information from a HTML webpage, one must understand what is HTML and how a webpage is designed in HTML.

HTML (HyperText Markup Language) is a markup language used to design the webpages. The web browsers read HTML files and render them into visible or audible webpages. There are many different tags in HTML and every tag has its own unique functionality. The body of a HTML webpage comes under the "body" tag of HTML, thus a web-scraper initially goes looking into a "body" tag and search for the keyword. And once that keyword is found a web-scraper checks the parent tag of that keyword and scrapes out everything written in that tag. This is how a web-scraper works and the bot uses it to get the information from the webpages dynamically.

```
from bs4 import BeautifulSoup
import requests
#Scraping
def scraper():
    result = ''
    r  = requests.get('http://www.example.com') #url to be scraped
    data = r.text
    soup = BeautifulSoup(data)                  #Scraper
    print 'Search Results'
    for i in soup.findAll('a'):
        if i.parent.name == 'p':                #scraping the text out of every
            print i                             #'p' tag in the webpage
scraper()
```

Fig. 11. Python implementation of a simple web-scraper to scrape out the text from 'p' tags of a url. BeatifulSoup is a library used for web-scraping.

So, a web crawler gives the links to the pages which contains the information and the web-scraper scrapes out the information from those links. So, if we use web-crawler along with a web-scraper, it would save a lot of running time for a bot. The bot has its source, the internet, and web-crawler along with a web-scraper is a way of getting the information from the source just like a human gets his information by reading. The bot, then stores that information within his memory by the insert function as shown in the Python implementation of the Trie in Fig. 5 and Fig. 8. The bot first adds the command into his memory and adds the information in the speak node at the end of that command in his memory.

4 Conclusion

This bot keeps on learning the things the user asks him with some pre-defined commands already added in the bot's memory. This bot is a basic but a powerful personal assistant and can be used by anyone and any electronic appliance if it is connected to the internet. The space complexity of this bot is low because of the modified Trie structure used to design its memory thus it is an optimized bot. This modified Trie is called a "do-and-speak trie".

Since we are moving to the modern age where everything is now digitalized and we're connected to the internet wherever we go so we need an assistant or computer to do the work for us. This bot can be held in a USB device and can be plugged in wherever we go and we can start communicating with it. This might ease our work to a greater extent and even though if someone's not connected to the internet, it can still be used to know the things which are already known to him. It is a platform independent bot thus it can be used in any device.

5 References

[1] Bamford, Simeon (2012), A framework for approaches to transfer of a mind's substrate.

[2] Black, Paul E. (2009-11-16). "trie". Dictionary of Algorithms and Data Structures.National Institute of Standards and Technology. Archived from the original on 2015-04-11.

[3] Goebel, Randy and Mackworth, Alan and Poole, David (1998). Computational Intelligence: A Logical Approach. New York: Oxford University Press. ISBN 0-19-510270-3.

[4] HTML [Def. 1]. (n.d.). In Merriam Webster Online, Retrieved November 4, 2015, from http://www.merriam-webster.com/dictionary/HTML.

[5] Kobayashi, M. and Takeda, K. (2000). "Information retrieval on the web". ACM Computing Surveys (ACM Press) 32 (2): 144–173. doi:10.1145/358923.358934.

[6] Kurzweil, Ray (2005), The Singularity is Near, Viking Press

[7] Simpson, J., & Weiner, E. (1989). bot. Oxford English Dictionary. Oxford: Oxford University Press. Retrieved from http://www.oxforddictionaries.com/definition/english/bot