
ACM
Association for
Computing Machinery

Today's Webscraping Workshop

- Automating the Wikipedia race: We want to get from the wikipedia article for the ACM to the wikipedia article for UMD
  - Need to do this only by clicking on links present in the wikipedia page
  - Shortest path wins
- Involves:
  - Webscraping - scraping data from webpages
  - Search Algorithms - wikipedia can be thought of as a graph of pages so we can use graph search algorithms to find the shortest path

Today's Webscraping Workshop

- Webscraping - Will be done using beautifulsoup
  - Get the html code for a given link
  - Find all the links in that page
- Search algorithms
  - Figures out which link to scrape at each step
  - Depth first search - WebscraperDFS.py
  - Breadth First Search - WebscraperBFS.py
  - A* - WebscraperAStar.py
- All work as is but you'll want to customize them, and at the end you'll have your own custom wikipedia racer

Download instructions

- On the Github ReadMe
- https://github.com/The-ACM-UMD/Webscraper-Workshop

Code - Depth first search

```
# Define the web scraper function
def scrape(url, depth_left, path, filter_func, target, keywords):
```

```
# If the current URL matches the target, return the path of visited URLs
if url == target:
    print(f"path of length {len(path)} found: {path}")
    quit()

# If the depth limit is reached, stop the recursion
elif depth_left == 0:
    print("This is going nowhere\n")
    return
```

Code - Depth first Search

```
else:
    try:
        # Send a GET request to the URL with a timeout of 0.05 seconds
        response = requests.get(url, timeout=0.05)

        # If the response status code is not 200 (OK), print an error message and stop
        if response.status_code != 200:
            print(f"Error fetching {url}: Status code {response.status_code}")
            return
```

```
# Parse the HTML content of the page using BeautifulSoup
# BeautifulSoup has several webpage parsers that converts the contents of the webpage,
# In this case the raw html code, and converts it to an easy to reference and read object
# lxml and html5lib are also good options but html.parser is built into python and is good
# enough for our purposes
soup = BeautifulSoup(response.content, 'html.parser')
```

Code - Depth first Search

```python
# Print the current page title and URL
print(f"Visiting {title} at {url}\n")

# Find all anchor (<a>) tags with the href attribute to get links
links = soup.find_all('a', href=True)

# Create a list of full URLs by joining the base URL with relative links
hrefs = [urljoin(url, link['href']) for link in links]
```

```python
# Filter the links to only include valid Wikipedia article URLs
hrefs_filtered = filter(
    lambda x: filter_func(x, url, path, title),  # avoid visiting the same or closely related page
    hrefs
)

# Recursively visit each filtered link
for href in hrefs_filtered:
    print(f"Attempting to visit {href}\n")
    # Add the current link to the path and continue scraping with reduced depth
    return scrape(href, depth_left - 1, path + [href], filter_func, target)
```

What you should care about

```python
# Main block to start scraping
if __name__ == "__main__":

    # Starting URL for scraping
    start_url = 'https://en.wikipedia.org/wiki/Association_for_Computing_Machinery'
    target = "https://en.wikipedia.org/wiki/University_of_Maryland,_College_Park" # The target URL we are trying to reach

    url_queue = []
    visited_set = set() # A visited set to make sure we don't revisit old links

    # CHANGE THESE AS YOU PLEASE
    max_depth = 10 # Maximum recursion depth
    def filter_func(next_url,current_url,path,title):
        return next_url.startswith("https://en.wikipedia.org/wiki/") and not ((next_url in current_url) or (current_url in ne

    url_queue.append((start_url, max_depth, [start_url]))
    # Start the scraping process from the start_url with the specified maximum depth
    while len(url_queue) != 0:
        current_url, depth, path = url_queue.pop()
        scrape(current_url, depth, path, filter_func, target)
```

Hint: go to wikipedia and see what special characters are in links that just refer to the page itself or go to special content pages, make sure those characters are filtered out

Changes - Breadth First Search

```python
# global visited set and url queue
visited_set = set() # a set that contains all of the links we have visited so we don't revisit the same links
url_queue = [] # a queue containing the links we have to visit that the algorithm draws from at each step
```

```python
# Main block to start scraping
if __name__ == "__main__":

    # Starting URL for scraping
    start_url = 'https://en.wikipedia.org/wiki/Association_for_Computing_Machinery'
    target = "https://en.wikipedia.org/wiki/University_of_Maryland,_College_Park" # The target URL we are trying to reach

    # CHANGE THESE AS YOU PLEASE
    max_depth = 10 # Maximum recursion depth
    def filter_func(next_url,current_url,path,title):
        return next_url.startswith("https://en.wikipedia.org/wiki/") and not ((next_url in current_url) or (current_url in next_url))

    url_queue.append((start_url, max_depth, [start_url]))
    # Instead of recursively scraping we now keep adding the links we want to click to the queue and scrape each link from the queue
    while len(url_queue) != 0:
        current_url, depth, path = url_queue.pop()
        scrape(current_url, depth, path, filter_func, target)
```

Changes - Breadth First Search

```python
def scrape(url, depth_left, path, filter_func, target):
    """
    Recursively scrapes web pages starting from a given URL up to a specified depth,
    searching for a target URL and printing the path if found. Otherwise it scrapes the page for all links
    of interest based on the filter argument and adds them to a queue, links are later picked from this queue
    in the "main" function to scrape again.
    Args:
        url (str): The starting URL to scrape.
        depth_left (int): The remaining depth to continue scraping.
        path (list): The list of URLs visited so far.
        filter_func (function): A function to filter valid URLs to visit.
        target (str): The target URL to find.
    Returns:
        None
    """
    # Check if page has been visited and if not add it to the visited set
    if url in visited_set:
        return

    # If the current URL matches the target, return the path of visited URLs
    if url == target:
        print(f"path of length {len(path)} found: {path}")
        quit()

    # If the depth limit is reached, stop the recursion
    elif depth_left == 0:
        print("This is going nowhere\n")
        return
```

Changes - Breadth First Search

```
# If the response status code is not 200 (OK), print an error message and stop
if response.status_code != 200:
    print(f"Error fetching {url}: Status code {response.status_code}")
    return

# If everything so far has worked we can count the page as visited to prevent a revisit
visited_set.add(url)

# Parse the HTML content of the page using BeautifulSoup
# BeautifulSoup has several webpage parsers that converts the contents of the webpage,
# In this case the raw html code, and converts it to an easy to reference and read object
# lxml and html5lib are also good options but html.parser is built into python and is good
# enough for our purposes
soup = BeautifulSoup(response.content, 'html.parser')
```

Changes - Breadth First Search

```
    # Filter the links to only include valid Wikipedia article URLs
    hrefs_filtered = filter(
        lambda x: filter_func(x, url, path, title),  # avoid visiting the same or closely
        hrefs
    )

    # Add each filtered link, the path to get there and the depth remaining to the queue
    for href in hrefs_filtered:
        # Add the current link to the path and continue scraping with reduced depth
        url_queue.append((href, depth-1, path + [href]))

# Handle any exceptions that occur during the request or scraping process
except Exception as e:
    print(f"Error fetching {url}: {e}")
```

Changes - A*

```python
# Main block to start scraping
if __name__ == "__main__":

    # Starting URL for scraping
    start_url = 'https://en.wikipedia.org/wiki/Association_for_Computing_Machinery'
    target = "https://en.wikipedia.org/wiki/University_of_Maryland,_College_Park" # The target URL we are trying to reach

    # CHANGE THESE AS YOU PLEASE
    max_depth = 10 # Maximum recursion depth
    def filter_func(next_url,current_url,path,title):
        return next_url.startswith("https://en.wikipedia.org/wiki/") and not ((next_url in current_url) or (current_url in n
    keywords = [] # These are the keywords the heuristic score is based on, the number of keywords found in the link = the h
    def heuristic_score(keyword):
        return 1 # The heuristic score for every keyword

    heapq.heappush(url_minheap,(0,(start_url, max_depth, [start_url])))
    # Start the scraping process from the start_url with the specified maximum depth
    while len(url_minheap) != 0:
        ( _ , (current_url, depth, path)) = heapq.heappop(url_minheap)
        scrape(current_url, depth, path, filter_func, target, keywords)
```

WHAT YOU SHOULD CARE ABOUT

```python
import heapq

# global visited link set and url minheap (replacing the url queue)
url_minheap = [] # A minimum heap with the links with the best chance of finding the shortest path at
```

Changes - A*

```
def scrape(url, depth_left, path, filter_func, target, keywords):
```

```
# Recursively visit each filtered link
for href in hrefs_filtered:
    # Add the current link to the path and continue scraping with reduced depth
    heuristic = 0
    for keyword in keywords:
        if keyword in href.split():
            heuristic += heuristic_score(keyword)

    heapq.heappush(url_minheap,(((len(path) + 3 - heuristic)),(href, depth-1, path + [href
```

Heuristic

- Prioritize every link based on what we think the path length that link will give us us is
- Path length = length of path so far + length of path left
  - Don’t actually know what length of the path left, let’s guess 3
  - But we do know that the more keywords we find in the text of the url the less of a path length we can expect
  - For example if we find ‘university’ or ‘maryland’ in the link text we know that there’s a lower expected path left
  - So our expected length of path becomes (3 - the number of keywords found)
    - 3 could be any number, but for ease of code and because I’m an optimist I picked 0
    - Can now add different values for each keyword

Changes - A*

```python
# Main block to start scraping
if __name__ == "__main__":

    # Starting URL for scraping
    start_url = 'https://en.wikipedia.org/wiki/Association_for_Computing_Machinery'
    target = "https://en.wikipedia.org/wiki/University_of_Maryland,_College_Park" # The target URL we are trying to reach

    # Maximum recursion depth
    max_depth = 10

    url_minheap = []
    visited_set = set() # A visited set to make sure we don't revisit old links

    # CHANGE THESE AS YOU PLEASE
    def filter_func(next_url,current_url,path,title):
        return next_url.startswith("https://en.wikipedia.org/wiki/") and not ((next_url in current_url) or (current_url in next_url))
    keywords = []
    def heuristic_score(keyword):
        return 1

    heapq.heappush(url_minheap,(0,(start_url, max_depth, [start_url])))
    # Start the scraping process from the start_url with the specified maximum depth
    while len(url_minheap) != 0:
        ( _ , (current_url, depth, path)) = heapq.heappop(url_minheap)
        scrape(current_url, depth, path, filter_func, target, keywords)
```

(Can use switch case on keywords for different scores)