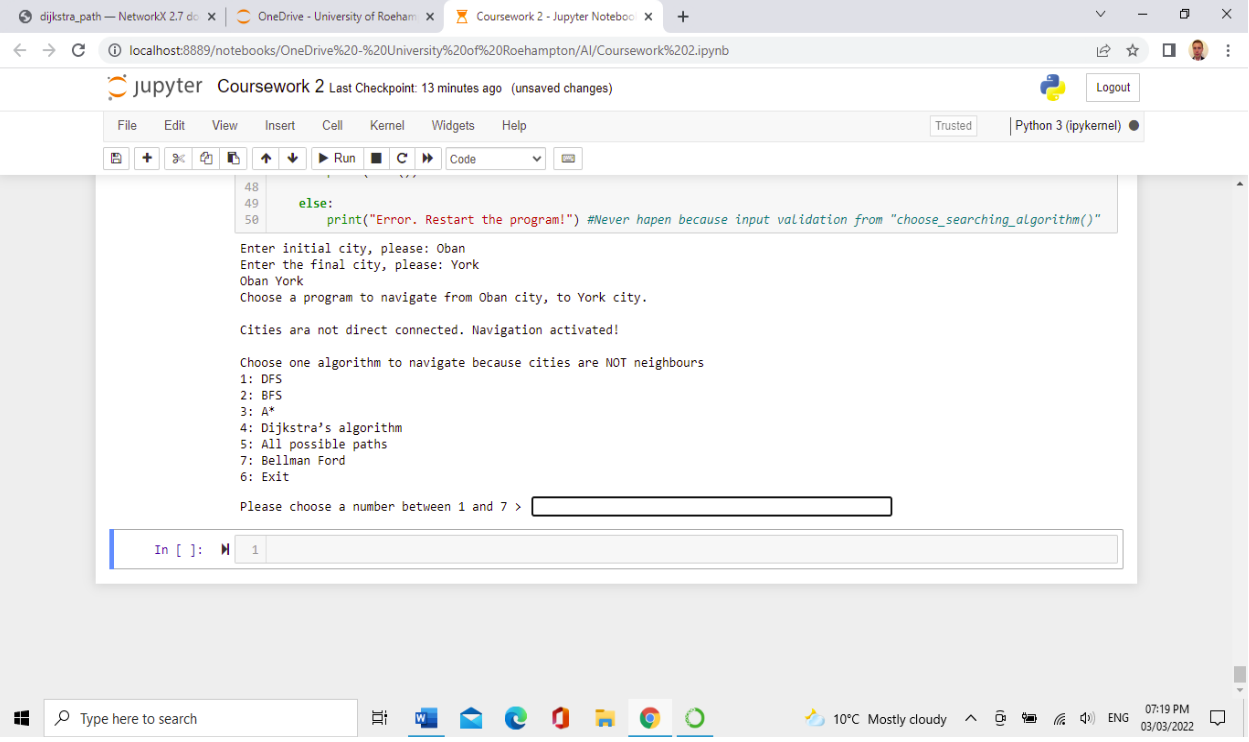
Coursework 2
“Deliveries”, version 1.0

YIT19488399, Tony.
Delivery date: 09 March 2022
Value: 25% of total

• A brief explanation of how you designed your navigation program. If you add a 4th algorithm, you should explain why you choose this algorithm.

This program is designed using only functions with a driving block code at the end. The program called “Deliveries” is allowing the user first to choose the initial city and the destination city, and secondly choose an algorithm to travel between cities. There are 5 algorithms in total designed in different block functions. Plus, there is an option to see all the possible routes between the two cities in two different formats. The algorithms are DFS, BFS, A*, Dijkstra and Bellman-Ford.

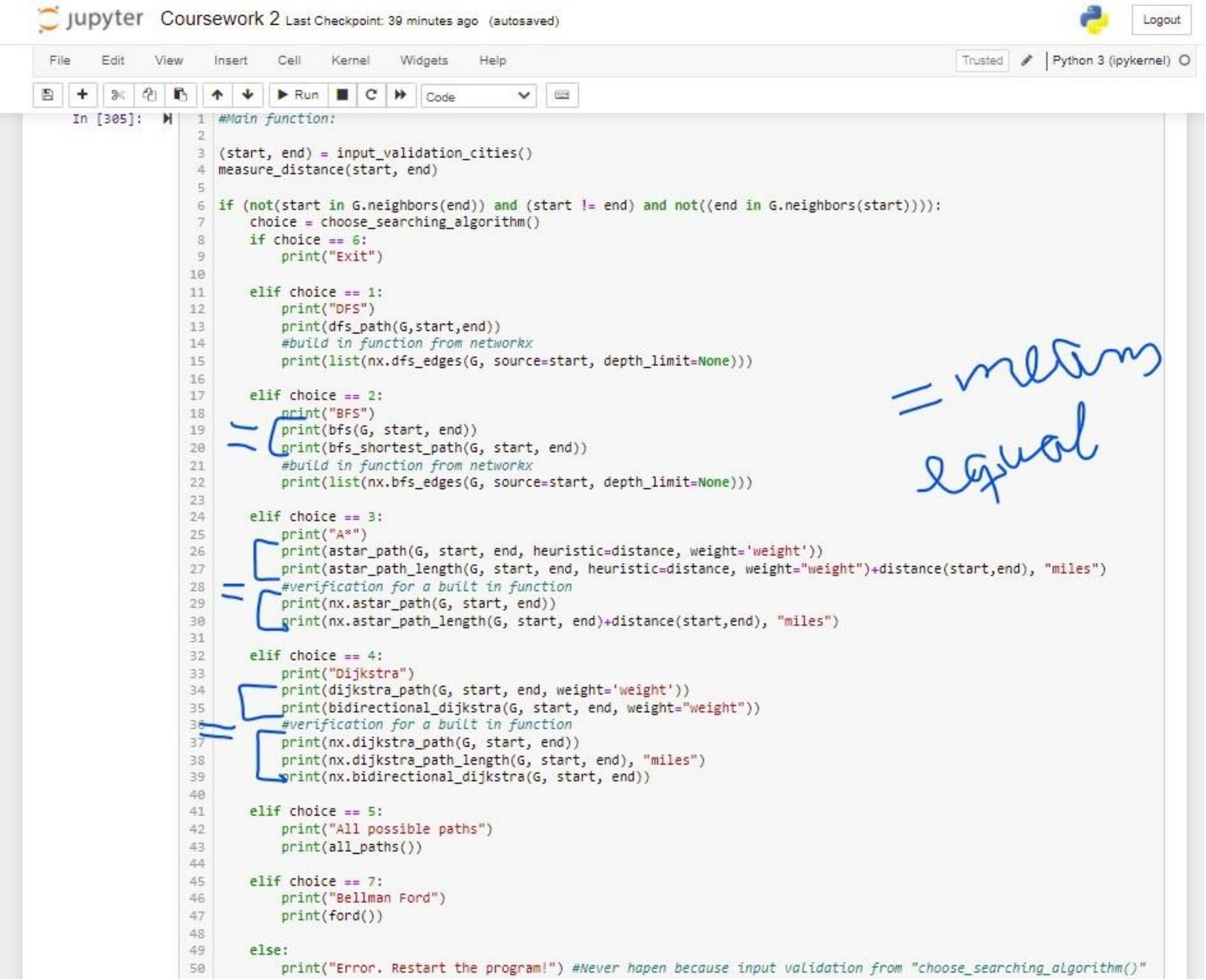


dijkstra_path — NetworkX 2.7 do
OneDrive - University of Roeham
Coursework 2 - Jupyter Noteboo
localhost:8889/notebooks/OneDrive%20-%20University%20of%20Roehampton/AI/Coursework%202.ipynb
jupyter Coursework 2 Last Checkpoint: 13 minutes ago (unsaved changes)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code
In [ ]:
Type here to search
10°C Mostly cloudy
ENG
07:19 PM
03/03/2022

```python

    else:
        print("Error. Restart the program!") #Never hapen because input validation from "choose_searching_algorithm()"
```

```
Enter initial city, please: Oban
Enter the final city, please: York
Oban York
Choose a program to navigate from Oban city, to York city.

Cities ara not direct connected. Navigation activated!

Choose one algorithm to navigate because cities are NOT neighbours
1: DFS
2: BFS
3: A*
4: Dijkstra’s algorithm
5: All possible paths
7: Bellman Ford
6: Exit

Please choose a number between 1 and 7 >
```

DFS, BFS and A* are explained in lectures with Dr Gu and Dr Arturo. Dijkstra's algorithm was explained by Dr Roberto in the previous semester this academic year. And Bellman-Ford algorithm is like Dijkstra, only allowing negative values of edges that do not exist in the present graph. A* algorithm uses and heuristic value and this is the only difference in between with Dijkstra's algorithm. BFS start from top to button and DFS from one side to the other side (left, right). I included some links commented in the source code in Jupyter Notebook.
I discovered while typing and doing my research about an interesting library called "Networkx" [https://networkx.org/, https://networkx.org/documentation/stable/index.html#]. I start developing my DFS and BFS algorithms and the next algorithms is a combination of long source code and build-in functions. The final block of code is just the build-in function. In the driving block, I combine both to verify that everything is the same, and this confirms that all is OK.

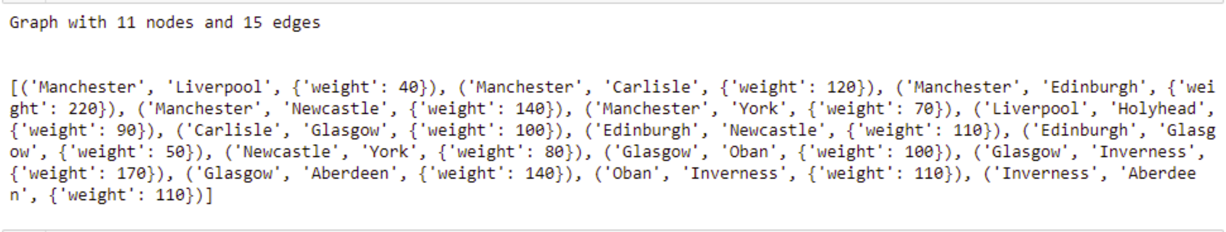


jupyter
Coursework 2 Last Checkpoint: 39 minutes ago (autosaved)
Logout
File Edit View Insert Cell Kernel Widgets Help
Trusted
Python 3 (ipykernel)
Run
Code
In [305]:

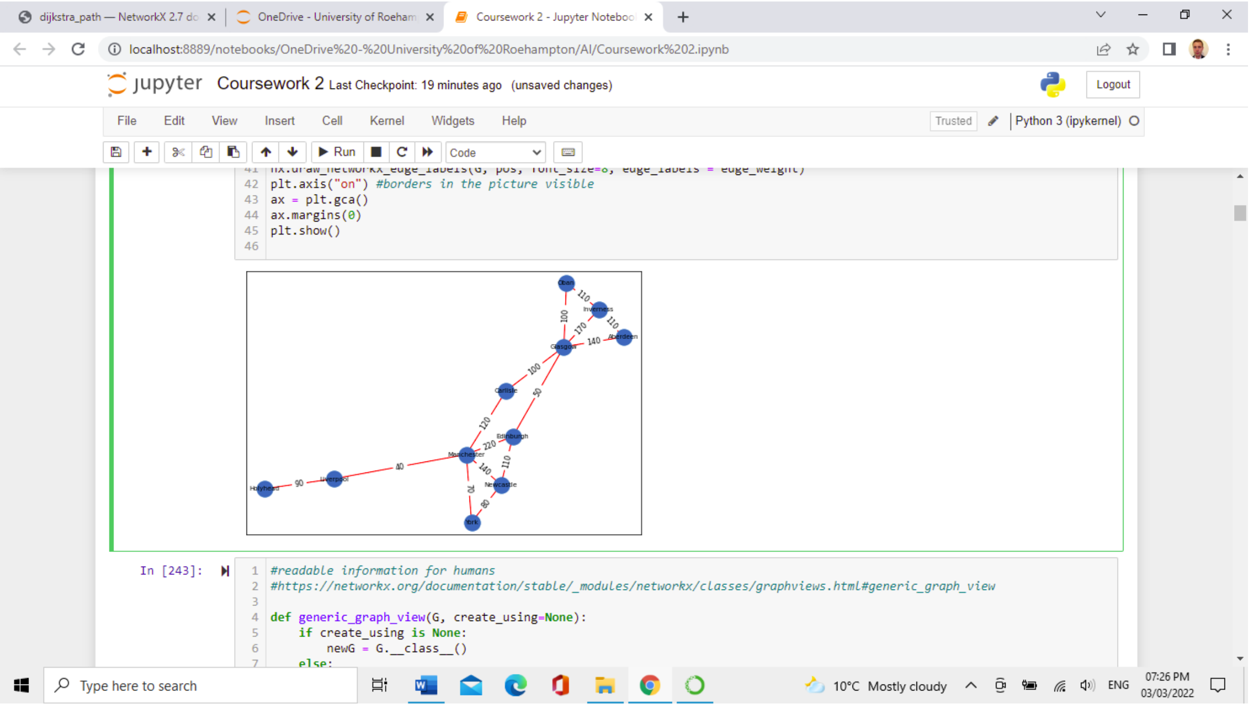
```python
#Main function:

(start, end) = input_validation_cities()
measure_distance(start, end)

if (not(start in G.neighbors(end)) and (start != end) and not((end in G.neighbors(start)))):
    choice = choose_searching_algorithm()
    if choice == 6:
        print("Exit")

    elif choice == 1:
        print("DFS")
        print(dfs_path(G,start,end))
        #build in function from networkx
        print(list(nx.dfs_edges(G, source=start, depth_limit=None)))

    elif choice == 2:
        print("BFS")
        print(bfs(G, start, end))
        print(bfs_shortest_path(G, start, end))
        #build in function from networkx
        print(list(nx.bfs_edges(G, source=start, depth_limit=None)))

    elif choice == 3:
        print("A*")
        print(astar_path(G, start, end, heuristic=distance, weight='weight'))
        print(astar_path_length(G, start, end, heuristic=distance, weight="weight")+distance(start,end), "miles")
        #verification for a built in function
        print(nx.astar_path(G, start, end))
        print(nx.astar_path_length(G, start, end)+distance(start,end), "miles")

    elif choice == 4:
        print("Dijkstra")
        print(dijkstra_path(G, start, end, weight='weight'))
        print(bidirectional_dijkstra(G, start, end, weight="weight"))
        #verification for a built in function
        print(nx.dijkstra_path(G, start, end))
        print(nx.dijkstra_path_length(G, start, end), "miles")
        print(nx.bidirectional_dijkstra(G, start, end))

    elif choice == 5:
        print("All possible paths")
        print(all_paths())

    elif choice == 7:
        print("Bellman Ford")
        print(ford())

    else:
        print("Error. Restart the program!") #Never hapen because input validation from "choose_searching_algorithm()"
```

= means equal

The beginning of my "Deliveries" program is to print a nice map of the UK. It has names and the distance between the cities. After I just developed code to confirm the correctness of my code and print a text with important and valued information.



```
Graph with 11 nodes and 15 edges


[('Manchester', 'Liverpool', {'weight': 40}), ('Manchester', 'Carlisle', {'weight': 120}), ('Manchester', 'Edinburgh', {'wei
ght': 220}), ('Manchester', 'Newcastle', {'weight': 140}), ('Manchester', 'York', {'weight': 70}), ('Liverpool', 'Holyhead',
{'weight': 90}), ('Carlisle', 'Glasgow', {'weight': 100}), ('Edinburgh', 'Newcastle', {'weight': 110}), ('Edinburgh', 'Glasg
ow', {'weight': 50}), ('Newcastle', 'York', {'weight': 80}), ('Glasgow', 'Oban', {'weight': 100}), ('Glasgow', 'Inverness',
{'weight': 170}), ('Glasgow', 'Aberdeen', {'weight': 140}), ('Oban', 'Inverness', {'weight': 110}), ('Inverness', 'Aberdee
n', {'weight': 110})]
```

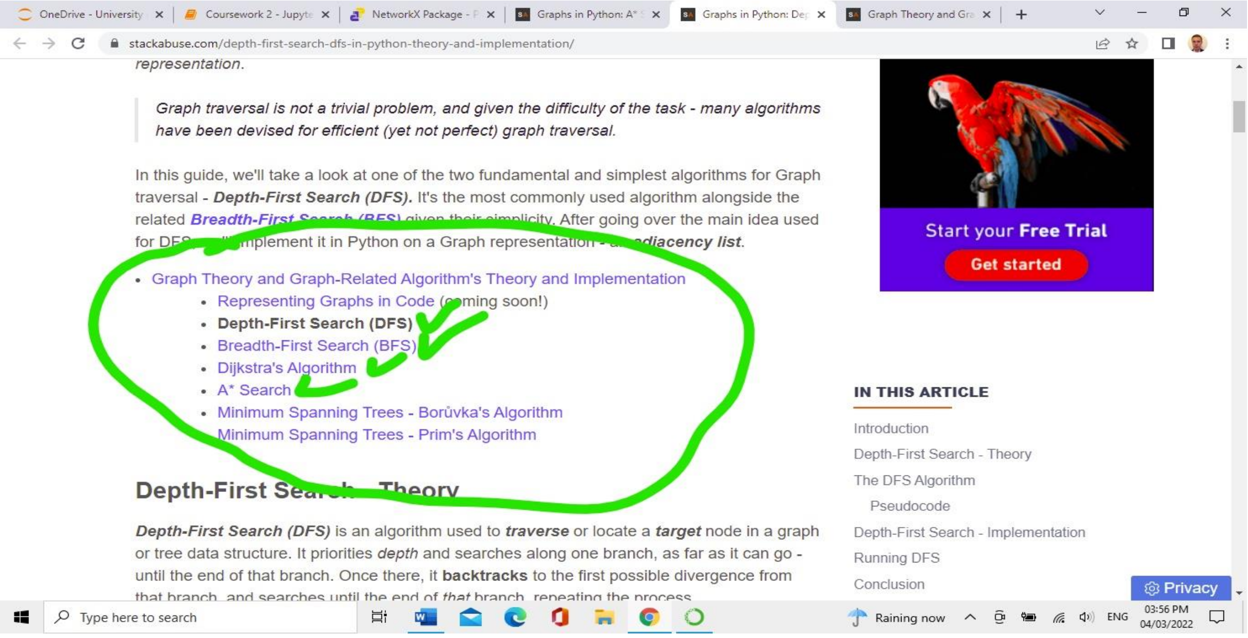


```
nx.draw_networkx_edge_labels(G, pos, font_size=8, edge_labels = edge_weight)
plt.axis("on") #borders in the picture visible
ax = plt.gca()
ax.margins(0)
plt.show()

```

```
#readable information for humans
#https://networkx.org/documentation/stable/_modules/networkx/classes/graphviews.html#generic_graph_view

def generic_graph_view(G, create_using=None):
    if create_using is None:
        newG = G.__class__()
    else:
```

Next blocks of code validate input from the user. Only allowed names for cities. After, I check if cities are the same, neighbours or not neighbours. If cities are not neighbours, function "choose_searching_algorithm()" ask for input an integer to choose an algorithm to travel between the cities in the UK map. And next blocks of code are the implementation of the searching algorithms presented at the beginning.

Resources used in this coursework are principally from the Internet:
https://www.askpython.com/python-modules/networkx-package to begin my graph design without knowledge about the "Networkx" library.
https://www.annytab.com/a-star-search-algorithm-in-python/ to understand one of the searching algorithms (A*).
https://stackabuse.com/graph-theory-and-graph-related-algorithms-theory-and-implementation/ and in general https://stackabuse.com/depth-first-search-dfs-in-python-theory-and-implementation/ helped me to understand and develop the rest of the searching algorithm in python.



representation.

> *Graph traversal is not a trivial problem, and given the difficulty of the task - many algorithms have been devised for efficient (yet not perfect) graph traversal.*

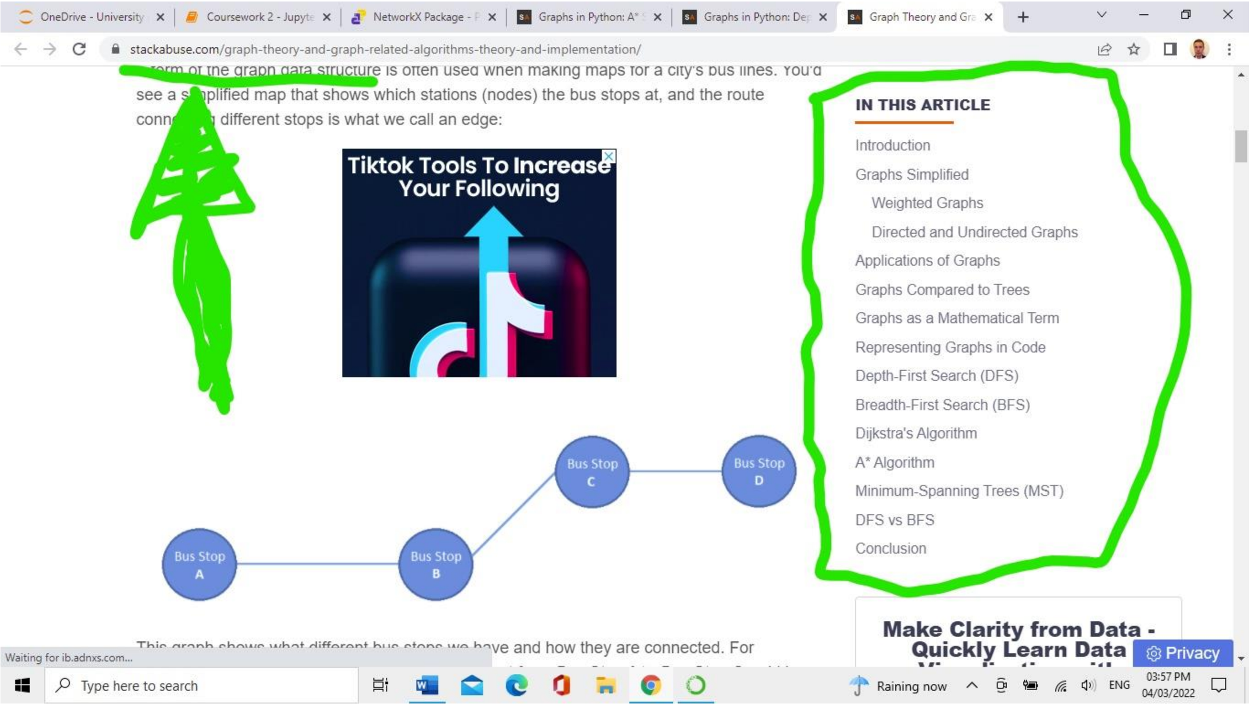
In this guide, we'll take a look at one of the two fundamental and simplest algorithms for Graph traversal - ***Depth-First Search (DFS).*** It's the most commonly used algorithm alongside the related ***Breadth-First Search (BFS)*** given their simplicity. After going over the main idea used for DFS, we'll implement it in Python on a Graph representation - ***an adjacency list***.

- Graph Theory and Graph-Related Algorithm's Theory and Implementation
  - Representing Graphs in Code (coming soon!)
  - **Depth-First Search (DFS)** ✓
  - Breadth-First Search (BFS) ✓
  - Dijkstra's Algorithm ✓
  - A* Search ✓
  - Minimum Spanning Trees - Borůvka's Algorithm
  - Minimum Spanning Trees - Prim's Algorithm

Depth-First Search - Theory

Depth-First Search (DFS) is an algorithm used to ***traverse*** or locate a ***target*** node in a graph or tree data structure. It priorities *depth* and searches along one branch, as far as it can go - until the end of that branch. Once there, it **backtracks** to the first possible divergence from that branch, and searches until the end of *that* branch, repeating the process.

IN THIS ARTICLE



...rm of the graph data structure is often used when making maps for a city's bus lines. You'd see a simplified map that shows which stations (nodes) the bus stops at, and the route connecting different stops is what we call an edge:

IN THIS ARTICLE

This graph shows what different bus stops we have and how they are connected. For

• Result of a demo run of your navigation program. You should take screenshots of your results from Jupyter Notebooks and insert them into your report.

This part of the coursework is included in the explanatory part of my program.

• A brief reflection. It may include but is not limited to the discussions about the topics below:

o Have you met any difficulties/bugs during this coursework, and how did you tackle them?

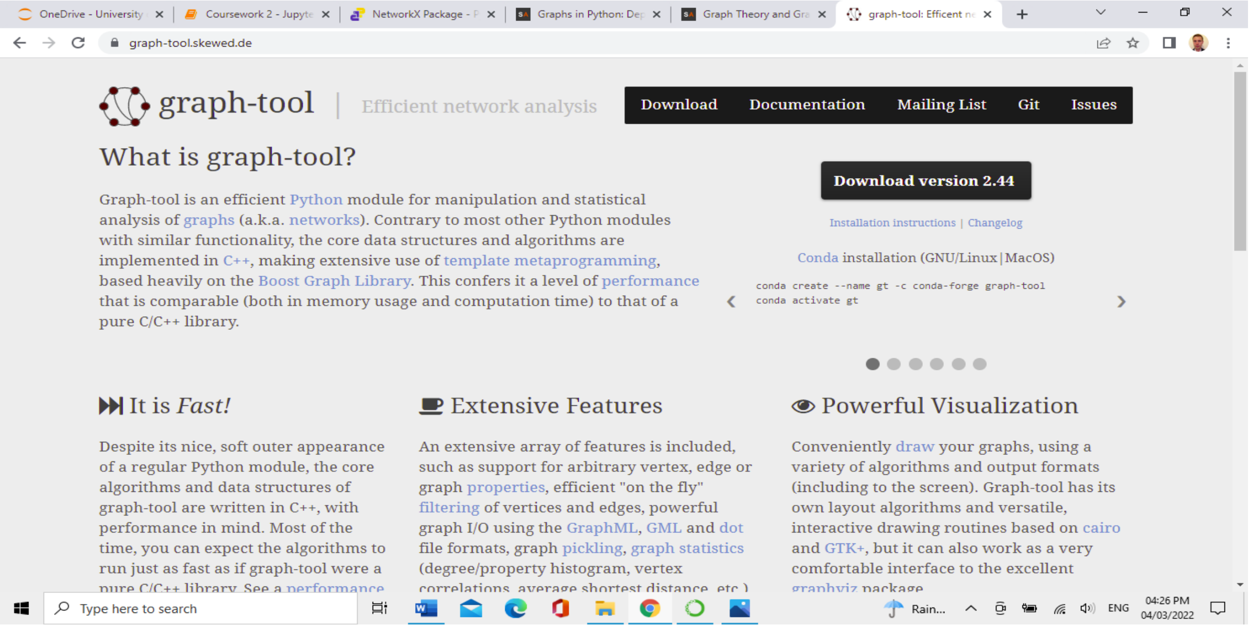
This coursework is easier than the previous, and the most difficult part was to design the A* searching algorithm because the heuristic function was difficult to define. I solved the problem by adding a heuristic value of the final node to the result.

o What did you think is the most difficult part of this coursework?

The most difficult part of this coursework is designing the searching algorithms.

o What have you learnt from investigating this problem?

I learned a new library that is very useful (“Networkx”). And I discovered another similar library but did not use it, because it is not popular and require installation. This last library is much advanced, complicated, and difficult to use. And I learned some build-in functions for searching algorithms in a weighted bidirected or undirected graph.



graph-tool | Efficient network analysis

What is graph-tool?

Graph-tool is an efficient Python module for manipulation and statistical analysis of graphs (a.k.a. networks). Contrary to most other Python modules with similar functionality, the core data structures and algorithms are implemented in C++, making extensive use of template metaprogramming, based heavily on the Boost Graph Library. This confers it a level of performance that is comparable (both in memory usage and computation time) to that of a pure C/C++ library.

Download version 2.44

Installation instructions | Changelog

Conda installation (GNU/Linux | MacOS)

```
conda create --name gt -c conda-forge graph-tool
conda activate gt
```

It is *Fast!*

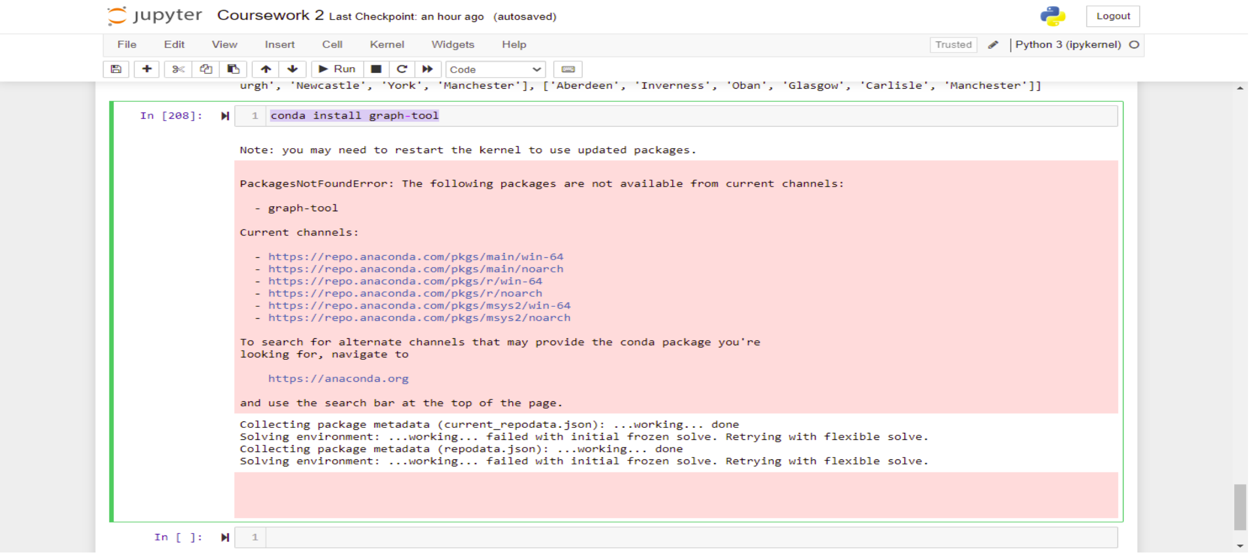
Despite its nice, soft outer appearance of a regular Python module, the core algorithms and data structures of graph-tool are written in C++, with performance in mind. Most of the time, you can expect the algorithms to run just as fast as if graph-tool were a pure C/C++ library. See a performance

Extensive Features

An extensive array of features is included, such as support for arbitrary vertex, edge or graph properties, efficient "on the fly" filtering of vertices and edges, powerful graph I/O using the GraphML, GML and dot file formats, graph pickling, graph statistics (degree/property histogram, vertex correlations, average shortest distance, etc.)

Powerful Visualization

Conveniently draw your graphs, using a variety of algorithms and output formats (including to the screen). Graph-tool has its own layout algorithms and versatile, interactive drawing routines based on cairo and GTK+, but it can also work as a very comfortable interface to the excellent graphviz package



```
urgh', 'Newcastle', 'York', 'Manchester'], ['Aberdeen', 'Inverness', 'Oban', 'Glasgow', 'Carlisle', 'Manchester']]
```

In [208]:

```
conda install graph-tool
```

```
Note: you may need to restart the kernel to use updated packages.

PackagesNotFoundError: The following packages are not available from current channels:

  - graph-tool

Current channels:

  - https://repo.anaconda.com/pkgs/main/win-64
  - https://repo.anaconda.com/pkgs/main/noarch
  - https://repo.anaconda.com/pkgs/r/win-64
  - https://repo.anaconda.com/pkgs/r/noarch
  - https://repo.anaconda.com/pkgs/msys2/win-64
  - https://repo.anaconda.com/pkgs/msys2/noarch

To search for alternate channels that may provide the conda package you're
looking for, navigate to

    https://anaconda.org

and use the search bar at the top of the page.

Collecting package metadata (current_repodata.json): ...working... done
Solving environment: ...working... failed with initial frozen solve. Retrying with flexible solve.
Collecting package metadata (repodata.json): ...working... done
Solving environment: ...working... failed with initial frozen solve. Retrying with flexible solve.
```

In [ ]:

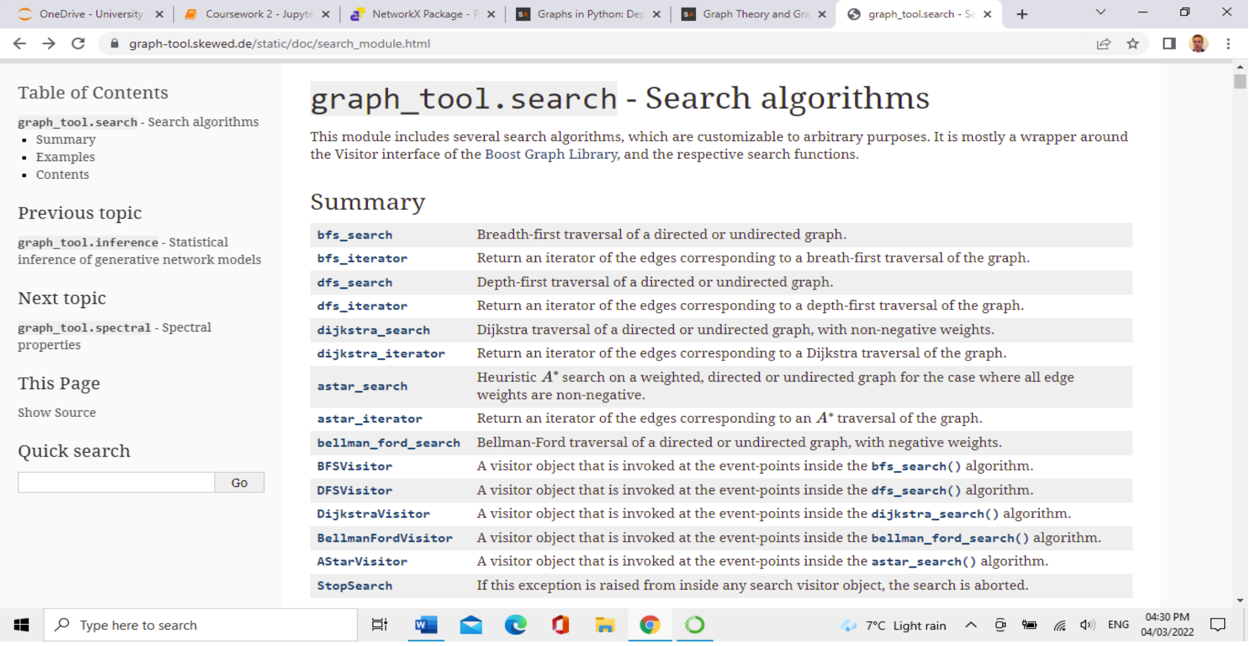


Table of Contents

- `graph_tool.search` - Search algorithms
  - Summary
  - Examples
  - Contents

Previous topic

`graph_tool.inference` - Statistical inference of generative network models

Next topic

`graph_tool.spectral` - Spectral properties

This Page

Show Source

Quick search

`graph_tool.search` - Search algorithms

This module includes several search algorithms, which are customizable to arbitrary purposes. It is mostly a wrapper around the Visitor interface of the Boost Graph Library, and the respective search functions.

Summary

`bfs_search`	Breadth-first traversal of a directed or undirected graph.
`bfs_iterator`	Return an iterator of the edges corresponding to a breath-first traversal of the graph.
`dfs_search`	Depth-first traversal of a directed or undirected graph.
`dfs_iterator`	Return an iterator of the edges corresponding to a depth-first traversal of the graph.
`dijkstra_search`	Dijkstra traversal of a directed or undirected graph, with non-negative weights.
`dijkstra_iterator`	Return an iterator of the edges corresponding to a Dijkstra traversal of the graph.
`astar_search`	Heuristic A^* search on a weighted, directed or undirected graph for the case where all edge weights are non-negative.
`astar_iterator`	Return an iterator of the edges corresponding to an A^* traversal of the graph.
`bellman_ford_search`	Bellman-Ford traversal of a directed or undirected graph, with negative weights.
`BFSVisitor`	A visitor object that is invoked at the event-points inside the **`bfs_search()`** algorithm.
`DFSVisitor`	A visitor object that is invoked at the event-points inside the **`dfs_search()`** algorithm.
`DijkstraVisitor`	A visitor object that is invoked at the event-points inside the **`dijkstra_search()`** algorithm.
`BellmanFordVisitor`	A visitor object that is invoked at the event-points inside the **`bellman_ford_search()`** algorithm.
`AStarVisitor`	A visitor object that is invoked at the event-points inside the **`astar_search()`** algorithm.
`StopSearch`	If this exception is raised from inside any search visitor object, the search is aborted.

o Which part(s) in your code design is your favourite and why?

My favourite part of the code is the design of the map because it shows a graphical output. I am very familiar with text output on the screen but less familiar with graphics.

o Which of the searching algorithms that you implemented is the best? Why?

Bellman-Ford is my preferred algorithm because is a build-in function that requires only 1 line of code.

o Any references you might have used to complete the task.

All the references are included in the previous text.

END.