CET 313 Artificial Intelligence

Workshop 2

Uninformed Search

Aims of the workshop

In the lecture, we introduce the concept of solving problems using uninformed search. Uninformed search algorithms are a type of search algorithm that does not use any knowledge about the problem being solved to guide the search. Instead, they explore the search space in a systematic way, visiting all possible nodes until they reach the goal node.

The three uninformed search algorithms that are covered in the workshop are:

1. Breadth-First Search (BFS): BFS explores the search space in a breadth-first manner, meaning that it visits all of the nodes at a given depth before moving on to the next depth. This can be implemented using a queue.
2. Depth-First Search (DFS): DFS explores the search space in a depth-first manner, meaning that it follows a single path until it reaches a dead end, and then backtracks to try a different path. This can be implemented using a stack.
3. Depth Limited DFS (DLS): DLS is a variant of DFS that limits the depth of the search. This could be useful for preventing DFS from getting stuck in deep, dead-end paths.

All three of these algorithms can be implemented in Python using the following basic steps:

1. Define a data structure to represent the search space. This could be a graph, a tree, or any other type of data structure that can be traversed.
2. Define a function to generate the successors of a given node. This function should return a list of all of the nodes that can be reached from the given node in a single step.
3. Define a function to check if a given node is the goal node. This function should return True if the given node is the goal node, and False otherwise.
4. Implement the search algorithm itself. This will typically involve using a queue or stack to keep track of the nodes that need to be explored.

Feel free to discuss your work with peers, or with any member of the teaching staff.

Reminder

We encourage you to discuss the content of the workshop with the delivery team and any findings you gather from the session.

Workshops are not isolated, if you have questions from previous weeks, or lecture content, please come and talk to us.

Exercises herein represent an example of what to do; feel free to expand upon this.

Helpful Resources

The programming language that we will be using is Python which is a great for learning AI basics. In this workshop we will be using functions, lists, conditions and any other basic python commands. If you would like a refresh on these concepts, you can check out one of the following tutorials:

- **Overview of Python** http://www.tutorialspoint.com/python/python_quick_guide.html
- **Good starting course** https://www.tutorialspoint.com/python/python_overview.htm
- Python Online Interactive http://www.learnpython.org/en/Hello%2C_World%21
- Python Hard Way http://learnpythonthehardway.org/book/ex20.html
- Python http://anh.cs.luc.edu/331/notes/PythonBasics.pdf
- Python http://learnpythonthehardway.org/book/ex20.html

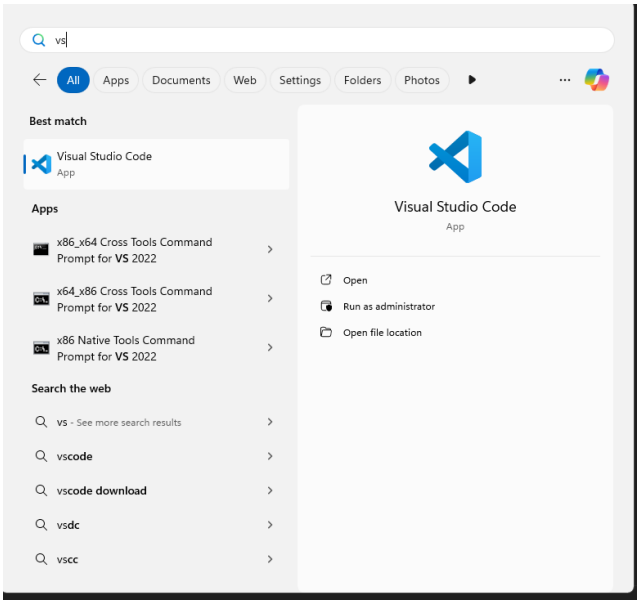
The 2 courses in bold will be particularly useful if you are new to this coding environment.

To write the scripts we will be using Jupyter notebooks, which is a helpful IDE for writing the scripts. To download it, you can use the link below:

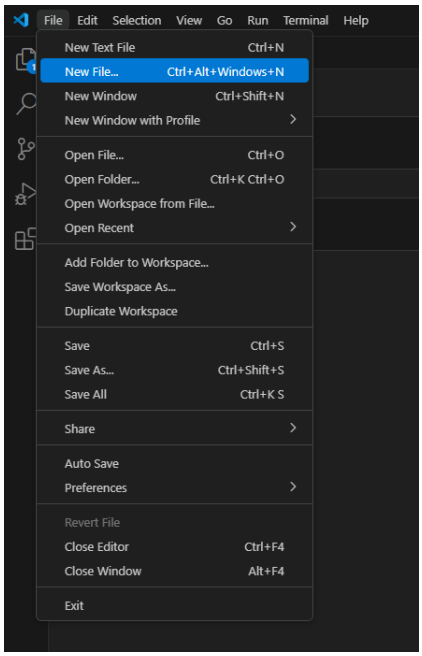
https://realpython.com/jupyter-notebook-introduction/

Setting up the Jupyter notebook

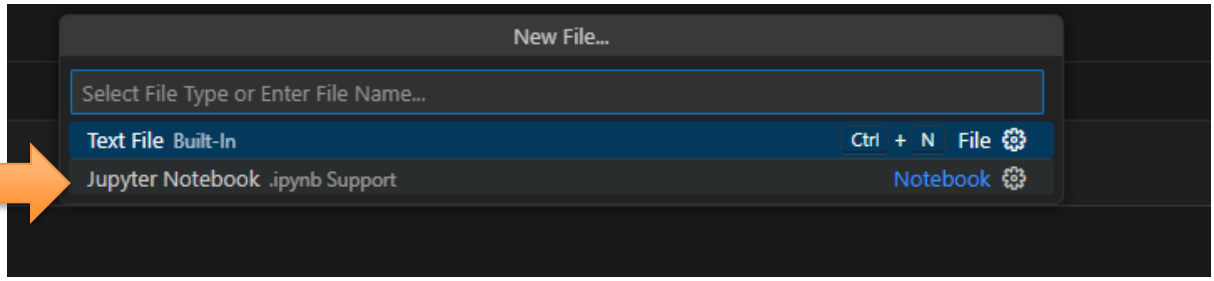
Launch VScode



Start a new file



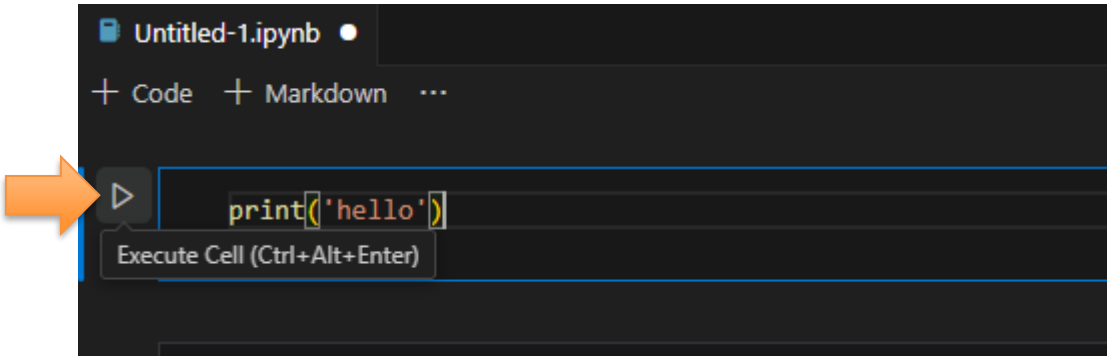
Choose Jupyter Notebook



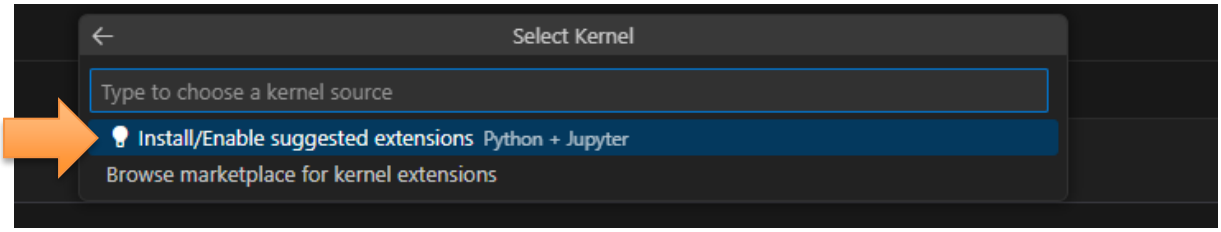
Type

```
print('Hello Word')
```

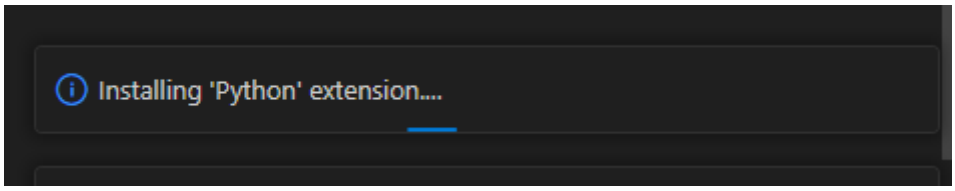
Then click Run



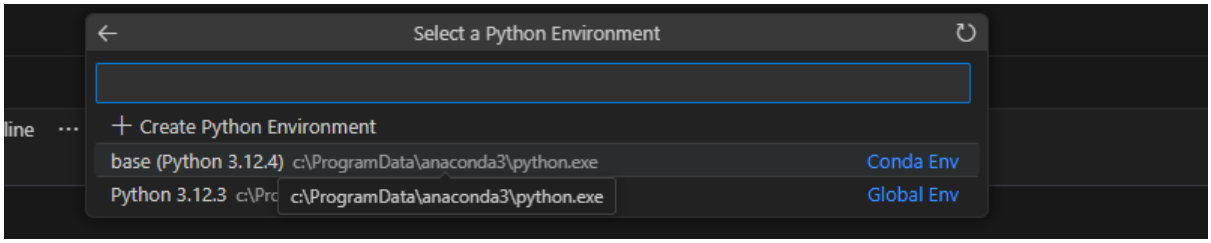
You will need to instal the extension



It will take few minutes to instal and show this notification



Run Hello Word again and choose base python



Now your PC is ready for programming 😊

Exercises

You may find it useful to keep track of your answers from workshops in a separate document, especially for any research tasks.
Where questions are asked of you, this is intended to make you think; it would be wise to write down your responses formally.

Exercise 1: We will start with this shown in the following graph:

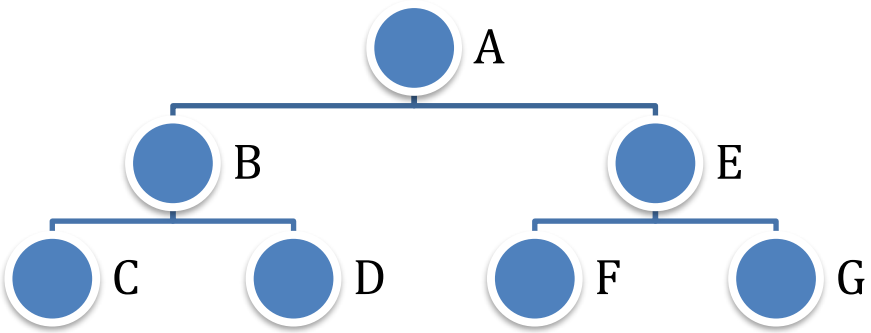


Fig. 1 Simple Problem Graph

Initially, we will start by defining the graph as a dictionary as follows:

```
graph = {
    'A': ['B', 'E'],
    'B': ['C', 'D'],
    'C':['C'],
    'D':['D'],
    'E': ['F', 'G'],
    'F':['F'],
    'G':['G']
}
```

Then we can define the BFS algorithm as a function that can be found below:

```
def bfs(graph, start, goal):
    visited = [] # Visited nodes list
    queue = [(start, [])] # Queue list

    while queue:
        node, path = queue.pop(0)  # Queue the first element
        print(f"Current node: {node}, Path: {path + [node]}")
        # Show the current node and path

        if node == goal: return path + [node]

        if node not in visited: visited.append(node)

        for neighbour in graph[node]:
                queue.append((neighbour, path + [node]))

    print("Goal not found")
    return None  # Goal not found
```

Finally, we can define the start and the goal nodes then call the function:

```
start_node = 'A'
goal_node = 'E'
result = bfs(graph, start_node, goal_node)
if result:
    print("Path found:", result)
else:
    print("No path found.")
```

To clarify the previous script let us trace it:

1. The bfs() function takes three arguments:
    - graph: A dictionary representing the graph. The keys of the dictionary are the nodes in the graph, and the values are lists of the neighbouring nodes of each key.
    - start: The starting node.
    - goal: The goal node.
2. The function initialises a list called visited to keep track of the nodes that have already been visited. It also initialises a list named queue with the starting node.
3. While the queue is not empty, the function removes the first node from the queue and checks if it is the goal node. If it is, the function returns a list of nodes representing the path from the start node to the goal node.
4. If the node is not the goal node, the function checks if it has already been visited. If it has not, the function marks it as visited and adds all of its neighbours to the queue.
5. If the queue is empty and the goal node has not been found, the function returns None.

Exercise 2: Experiment with three different starting and ending points and observe the problem solution.

Exercise 3: (Challenge 🧠) Use the time library to calculate how long it would take to solve the problem using three different starting points and ending points
Hint: you can refer to this tutorial to get you started:
https://realpython.com/python-time-module/

Exercise 4: (Challenge 🧠) Change the script to mode the following graph:

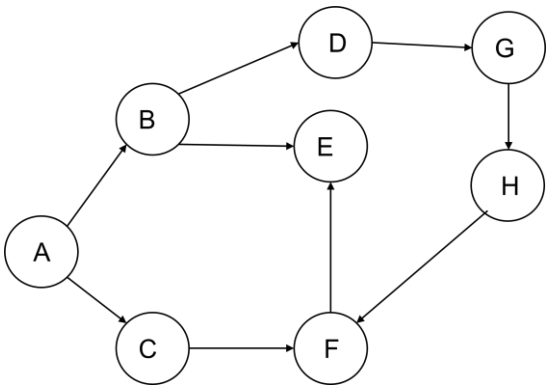


Fig. 2 Second Problem Graph

Exercise 5: Update the script to find the solution to the problem and experiment with different starting points, e.g., start from A and end at H

Exercise 6: In the following script we will use DFS to solve the simple graph. The DFS algorithm is written as a function as follows:

```
def dfs(graph, start, goal):
    visited = []  # Visited nodes list
    stack = [(start, [])]  # Stack for DFS (similar to the queue in BFS)

    while stack:
        node, path = stack.pop()  # Pop the last element (LIFO for DFS)
        print(f"Current node: {node}, Path: {path + [node]}")
        # Show the current node and path

        if node == goal: return path + [node]

        if node not in visited: visited.append(node)
# Iterate through neighbours in reverse to maintain order like recursion
            for neighbour in reversed(graph[node]):
                stack.append((neighbour, path + [node]))

    print("Goal not found")
    return None  # Goal not found
```

Copy the script and use this function to solve the problem in Figure 1.

Exercise 7: Update the graph to use the problem defined in Figure 2.

Exercise 8: Experiment with 3 different starting and ending points and observe the problem solution.

Exercise 9: **(Challenge 🧠)** write a code to compare the time to find the solution using both algorithms and graphs**.**

Exercise 10: In this exercise we will be using DLS algorithm to solve the problem, use the following script to implement it:

```
def dls(graph, start, goal, limit):
    visited = []  # Visited nodes list
    stack = [(start, [], 0)]
    # Stack for DFS (node, path, current depth)

    while stack:
        node, path, depth = stack.pop()
        # Pop the last element (LIFO for DFS)
        print(f"Current node: {node},Path:{path+[node]}, Depth:{depth}")
        # Show the current node, path, and depth

        if node == goal:
            return path + [node]

        if node not in visited:
            visited.append(node)

            if depth < limit:  # Only expand if depth is below the limit
            # Iterate through neighbours in reverse to maintain DFS behaviour
                for neighbour in reversed(graph[node]):
                    stack.append((neighbour, path + [node], depth + 1))

    print(f"Goal not found within depth limit {limit}")
    return None  # Goal not found within the depth limit
```

Copy the script and use this function to solve the problem in Figure 1.

Exercise 11: Update the graph to use the problem defined in Figure 2.

Exercise 12: Experiment with 3 different starting and ending points and observe the problem solution.

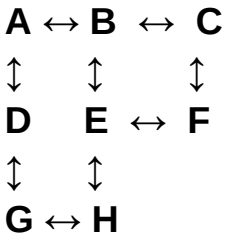
Exercise 13: Change the max depth and find how can use break the algorithm

Exercise 14: Make your own graph that has many many layers and do a comparison of the different algorithms

Exercise 15: Write a script that will take the number of levels the tree has and the number of nodes that branches from the nodes.

Exercise 16: Solve the new graph using the 3 algorithms

Exercise 17: (Challenge 🧠) In this exercise we will be using search algorithms to solve a maze problem. Let us start by defining the following maze:



Write a script to define the maze as a dictionary as done in exercise 1.

Exercise 17: (Challenge 🧠) Use DFS algorithm to solve the maze.

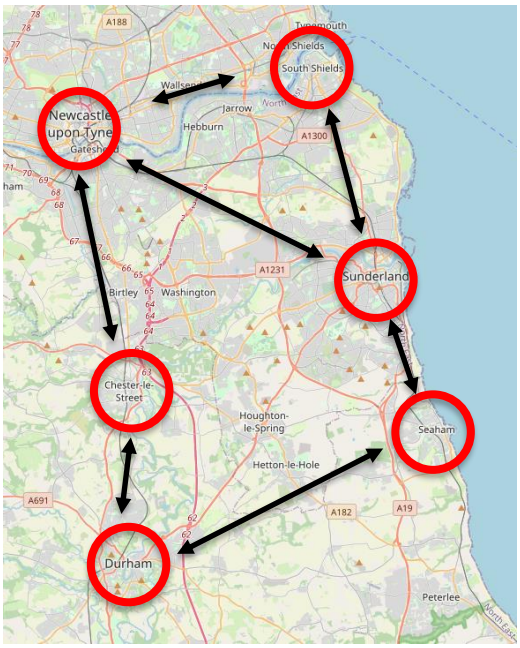
Exercise 18: (Challenge 🧠) Use BFS algorithm to solve the maze.

Exercise 19: (Challenge 🧠) Use DLS algorithm to solve the maze.

Exercise 20: (Challenge 🧠 🧠) Do a comparison of both algorithms BFS and DFS and find the number of nodes needed to solve the problem.

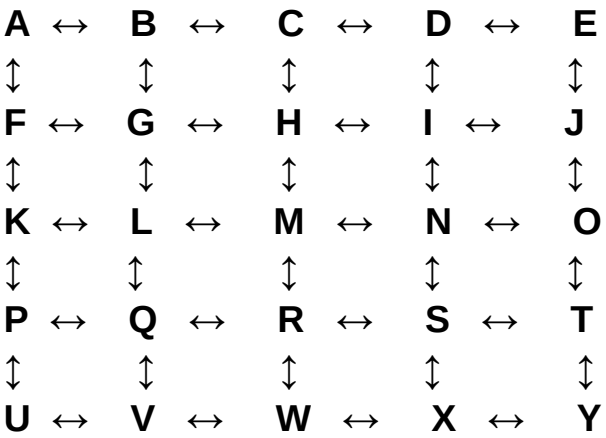
Exercise 21: (Challenge 🧠) Try different starting and ending points then calculate the number of steps needed to solve the problem.

Exercise 22: (Challenge 🧠) Change the maze map to make represent this maze:



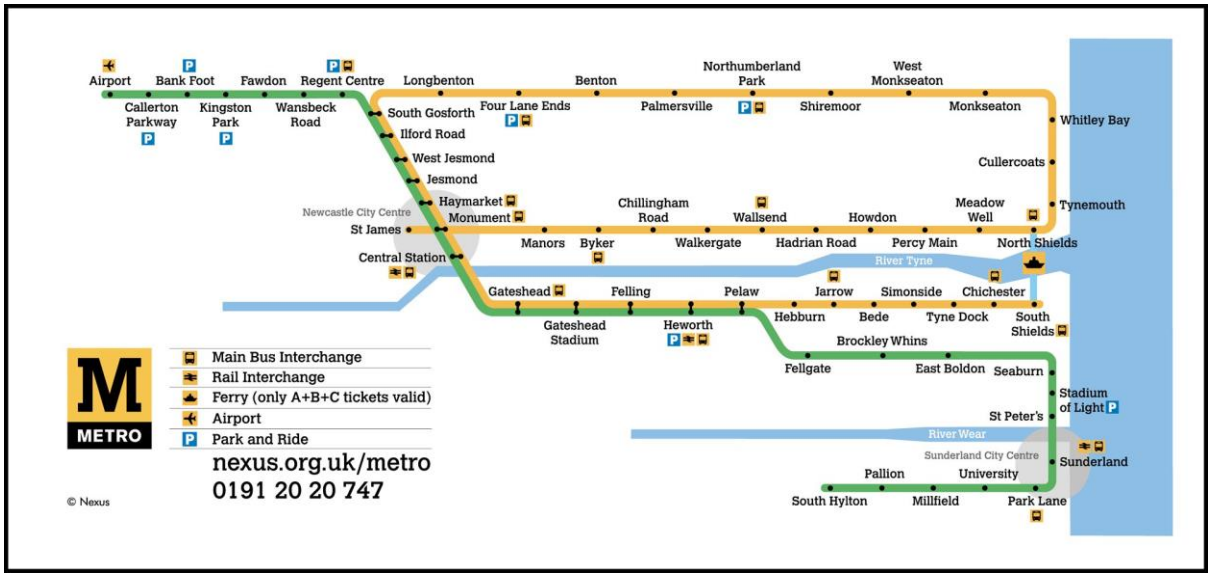
and solve it using DFS, BFS and DLS, to find the route between Sunderland and all other highlighted locations.

Exercise 22: (Challenge 🧠) Change the maze map to make represent this maze:



and solve it using DFS, BFS and DLS.

Exercise 23: (Challenge 🧠 🧠 🧠) In this exercise, use the BFS, and DFS to find the route for the Sunderland metro map shown in following Figure. Set the starting point to be Sunderland and the end point to be Whitley Bay.



https://www.nexus.org.uk/themes/custom/nexus/images/metro-map-large.jpg

Exercise 24: Experiment with different stating and ending points and double check if the route is correct using the map.

Exercise 25: (Challenge 🧠) Compare the number of stations the metro would pass through using each algorithm.

Make sure you upload your notebook with the solutions to your eportfoilio

END OF EXERCISES