Table of Contents

Objectives

After performing this lab, students shall be able to understand implementation of the following uninformed/blind search algorithms::

- ✓ BFS
- ✓ DFS
- ✓ UCS
- ✓ IDS

1 Blind Search Algorithms

Blind search algorithms (also called uninformed search algorithms) are search techniques that do not have any additional information about the problem space other than:

- The initial state
- The goal state
- The actions available to move between states

They explore the search space systematically without knowing:

- How close a state is to the goal
- Which path is better or worse

These algorithms work purely based on structure, not intelligence or heuristics.

1.1 State Space Representation

In blind search:

- Each state represents a configuration of the problem
- States are connected through actions
- The search space is usually represented as:
  - A tree, or
  - A graph

Each algorithm differs in how it explores this state space.

1.1.1 What is a "State" in Python?

A state is any data structure that uniquely represents a position in the problem.
Examples:

- A node name: 'A'
- A position: (x, y)
- A puzzle configuration: [[1,2,3],[4,5,6],[7,8,0]]

States should be hashable so they can be stored in sets or dictionaries.

1.1.2 Graph Representation in Python

Most blind search algorithms assume the problem is a graph.

The most common Python representation is an adjacency list:

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

If costs are involved:

```
graph = {
    'A': [('B', 1), ('C', 3)],
    'B': [('D', 2)],
    'C': [('F', 4)],
}
```

1.2 Node Representation

Search algorithms do not only store states.They store nodes. A node contains:

- Current state
- Parent node (to reconstruct the path)
- Depth (for IDS)
- Path cost (for UCS)

Typical Node Class in Python

```
class Node:
    def __init__(self, state, parent=None, cost=0, depth=0):
        self.state = state
        self.parent = parent
        self.cost = cost
        self.depth = depth
```

1.3 Types of Blind Search Algorithms

1.3.1 Breadth First Search (BFS)

Breadth First Search (BFS) explores a graph level by level, visiting all neighboring nodes before moving deeper. It uses a queue data structure and is commonly applied in problems where the shortest path in an unweighted graph is required. BFS is complete and guarantees the shortest path, but it requires significant memory.

- Create a queue
- Add initial node
- While queue is not empty:
    - Remove the first node
    - Check if it is the goal
    - Expand its neighbors
    - Add unexplored neighbors to the queue

1.3.2 Depth First Search (DFS)

Depth First Search (DFS) explores as far as possible along a branch before backtracking. It uses a stack or recursion and is memory efficient. DFS is useful for exploring large graphs, detecting cycles, and solving maze-like problems, although it does not guarantee the shortest path.

- Push initial node onto stack
- While stack is not empty:
    - Pop the top node
    - Check goal
    - Push children onto stack

1.3.3 Uniform Cost Search (UCS)

Uniform Cost Search (UCS) explores a graph by always expanding the node with the lowest cumulative path cost from the start node. Unlike Breadth First Search, which assumes all edges have equal cost, UCS is designed for graphs where edges may have different weights. It uses a priority queue, where nodes are ordered based on their total cost, ensuring that the least-cost path is explored first. UCS is both complete and optimal, meaning it will always find a solution if one exists and will guarantee the cheapest path to the goal. However, because it may need to explore many nodes and repeatedly manage the priority queue, it can be computationally expensive and require significant memory. UCS is commonly used in routing, navigation, and cost-based decision problems.

Priority Queue Behavior: Python's `heapq` sorts based on the first element of a tuple:

```
(cost, node)
```

- Push (0, start_node) into heap
- Pop lowest-cost node
- Expand neighbors
- Update costs if a cheaper path is found

1.3.4 Iterative Deepening Search (IDS)

Iterative Deepening Search (IDS) is a search algorithm that combines the advantages of Depth First Search and Breadth First Search. It performs a series of depth-limited DFS operations, starting with a depth limit of zero and gradually increasing the limit until the goal is found. By doing this, IDS avoids the high memory usage of BFS while still guaranteeing the discovery of the shallowest (shortest) solution, like BFS. IDS uses very little memory because, at any time, it behaves like DFS, storing only the nodes along the current path. This makes it both complete and optimal for unweighted graphs. Although IDS repeats some work by re-exploring nodes at shallow depths, the overall overhead is small, making it an efficient and practical choice when memory is limited.

IDS repeatedly applies Depth-Limited DFS.

- Depth limit = 0
- Then 1, 2, 3, …
- Stops when goal is found

IDS Components

- Depth-Limited DFS function
- Loop that increases depth limit

Depth-Limited DFS Condition

```
if node.depth > limit:
    return
```

In summary:

Algorithm	Data Structure	Optimal	Complete	Memory
BFS	Queue	Yes	Yes	High

DFS	Stack	No	No	Low
UCS	Priority Queue	Yes	Yes	High
IDS	Stack + Loop	Yes	Yes	Medium

2 Python Basics for Implementing Search Algorithms

2.1 Classes and Objects

Python is an object oriented programming language. Almost everything in Python is an object, with its properties and methods. A Class is like an object constructor, or a "blueprint" for creating objects.

2.1.1 Defining a Class in Python

Like function definitions begin with the `def` keyword in Python, class definitions begin with a `class` keyword. The first string inside the class is called docstring and has a brief description of the class. Although not mandatory, this is highly recommended. Here is a simple class definition,

```
class MyNewClass:
    '''This is a docstring. I have created a new class'''
    pass
```

A class creates a new local namespace where all its attributes are defined. Attributes may be data or functions. As soon as we define a class, a new class object is created with the same name. This class object allows us to access the different attributes as well as to instantiate new objects of that class.

```
class Person:

    "This is a person class"
    age = 10

    def greet(self):
        print('Hello')
```

```
# Output: 10
print(Person.age)

# Output: <function Person.greet>
print(Person.greet)

# Output: "This is a person class"
print(Person.__doc__)
```

2.1.2 Creating Objects

The procedure to create an object is similar to a function call.

```
harry = Person()
```

2.1.3 Constructors in Python

Class functions that begin with double underscore __ are called special functions as they have special meaning.
Of one particular interest is the `__init__()` function. This special function gets called whenever a new object of that class is instantiated.
This type of function is also called constructors in Object Oriented Programming (OOP).
We normally use it to initialize all the variables.

```
class ComplexNumber:
    def __init__ (self, r=0, i=0):
        self.real = r
        self.imag = i

def get_data(self):
    print(f'{self.real}+{self.imag}j')

# Create a new ComplexNumber object
num1 = ComplexNumber(2, 3)
```

We can even delete the object itself, using the del statement.

```
del c1
```

2.1.4 Inheritance in Python

Inheritance is a powerful feature in object oriented programming. It refers to defining a new class with little or no modification to an existing class. The new class is called derived (or child) class and the one from which it inherits is called the base (or parent) class.

```
class BaseClass:
    #Body of base class
class DerivedClass(BaseClass):
    #Body of derived class
```

2.2 Python Stacks

A stack is a data structure that stores items in a Last-In/First-Out manner. We will look at three different implementations of stacks in Python.

- Using list data structure
- Using collections.deque module

2.2.1 Stacks via Lists

The built-in Python list object can be used as a stack. For stack **.push()** method, we can use **.append()** method of list. **.pop()** method of list can remove elements in LIFO order. Popping an empty list (stack) will raise an IndexError. To get the top most item (peek) in the stack, write **list[-1]**. Bigger lists (stacks) often run into speed issues as they continue to grow. list may be familiar, but it should be avoided because it can potentially have memory reallocation issues.

```
myStack = []
myStack.append('a')
myStack.append('b')
myStack.append('c')
myStack # ['a', 'b', 'c']
myStack.pop()
myStack.pop()
myStack.pop()
# 'c'
# 'b'
# 'a'
```

2.2.2 Stacks via collections.deque

This method solves the speed problem we face in lists. The deque class has been designed as such to provide O(1) time complexity for append and pop operations. The deque class is built on top of a doubly linked list structure which provides faster insertion and removal. Popping an empty deque gives the same IndexError.

```
from collections import deque
myStack = deque()

myStack.append('a')
myStack.append('b')
myStack.append('c')
myStack # deque(['a', 'b', 'c'])
myStack.pop() # 'c'
myStack.pop() # 'b'
myStack.pop() # 'a'
```

2.3 Python Queues

A queue is FIFO data structure. The insert and delete operations are sometimes called enqueue and dequeue. We can use list as a queue as well. To follow FIFO, use **pop(0)** to remove the first element of the queue. But as discussed before, lists are slow. They are not ideal from performance perspective. We can use the **collections.deque** class again to implement Python queues. They work best for non-threaded programs. We can also use **queue.Queue** class. But it works well with synchronized programs. If you are not looking for parallel processing, **collections.deque** is a good default choice.

```
from collections import deque
q = deque()
q.append('eat')
q.append('sleep')
q.append('code')
q
# deque(['eat', 'sleep', 'code'])
q.popleft()
# 'eat'
```

```
q.popleft()
q.popleft()
# 'sleep'
# 'code'
q.popleft()
# IndexError: "pop from an empty deque"
```

2.4 Python Priority Queues

A priority queue is a special type of queue in which each element is associated with a priority value, and elements are removed based on their priority rather than their insertion order. Unlike normal queues (FIFO) or stacks (LIFO), a priority queue always removes the element with the highest or lowest priority first. In search algorithms such as Uniform Cost Search (UCS) and A*, priority queues are essential because nodes must be expanded based on minimum path cost instead of arrival time.

In Python, priority queues are most commonly implemented using the heapq module, which provides an efficient way to maintain a collection of elements in sorted order based on priority.

To use a priority queue in Python, elements are stored as **tuples**, where the first value represents the priority and the second value represents the actual data. The heap automatically compares the first element of the tuple to determine priority.

```
import heapq

pq = []

heapq.heappush(pq, (1, 'task A'))
heapq.heappush(pq, (3, 'task C'))
heapq.heappush(pq, (2, 'task B'))

pq
# [(1, 'task A'), (3, 'task C'), (2, 'task B')]

heapq.heappop(pq)
# (1, 'task A')

heapq.heappop(pq)
# (2, 'task B')
```

```
heapq.heappop(pq)
# (3, 'task C')
```

Even though the internal list may not appear sorted, the heap always guarantees that the smallest priority element is removed first.

2.5 Python File Handling

Python allows users to handle files by supporting to read and write files, along with many other file handling options.

2.5.1 Open & Close a file

When you want to read or write a file, the first thing to do is to open the file. Python has a built-in function open that opens the file and returns a file object. To return a file object we use open() function along with two arguments, that accepts file name and the mode, whether to read or write.

The syntax is given below:

open(filename, mode)

2.5.2 Kinds of modes

There are three basic types of modes in which files can be opened in Python.

mode	meaning
r	open for reading (default)
r+	open for both reading and writing (file pointer is at the beginning of the file)
w	open for writing (truncate the file if it exists)
w+	open for both reading and writing (truncate the file if it exists)
a	open for writing (append to the end of the file if exists & file pointer is at the end of the file)

Always keep in mind that the mode argument is not mandatory. If not passed, then Python will assume it to be " r " by default.

Let's look at this program and try to analyze how the read mode works:

```
# a file named "book", will be opened with the reading mode.
file = open('book.txt', 'r')
# This will print every line one by one in the file
for each in file:
    print (each)
```

2.5.3 Working of read() mode

There is more than one way to read a file in Python. If you need to extract a string that contains all characters in the file then we can use file.read(). The full code would work like this:

```
file = open("file.text", "r")
print (file.read())
```

Another way to read a file is to call a certain number of characters like in the following code the interpreter will read the first five characters of stored data and return it as a string:

```
# Python code to illustrate read() mode character wise
file = open("file.txt", "r")
print (file.read(5))
```

2.5.4 Working of write() mode

Let's see how to create a file and how write mode works:
To manipulate the file, write the following in your Python environment:

```
# Python code to create a file
file = open('book.txt','w')
file.write("This is the write command")
file.write("It allows us to write in a particular file")
file.close()
```

The close() command terminates all the resources in use and frees the system of this particular program.

2.5.4 Working of append() mode

```
# Python code to illustrate append() mode
file = open('book.txt','a')
file.write("This will add this line")
file.close()
```

Exercise Questions:

State Space & BFS (Breadth-First Search) — Shortest Path in a City Map
Question: 1
A city map is represented as a graph where intersections are nodes and roads are edges.
Write a Python program to:

- Represent the state space of the city as a graph (use adjacency list).
- Implement **Breadth-First Search (BFS)** using a **queue** to find the shortest path (minimum number of roads) from a given start location to a destination.
- Display:
  - Visited nodes order
  - Shortest path

Node Representation & DFS — Maze Solver
Question: 2
A maze is represented as a grid where:

- 0 = path
- 1 = wall

Write a Python program to:

- Represent each cell as a **node** with coordinates.
- Implement **Depth-First Search (DFS)** using a **stack** to find a path from start to exit.
- Print the path found.

Uniform Cost Search (UCS) — Delivery Route Optimization
Question: 3
A delivery company needs to find the **lowest cost route** between warehouses.

- Each road has a travel cost.
- Represent the graph with weighted edges.
- Implement **Uniform Cost Search (UCS)** using a **priority queue**.
- Output:
  - Minimum cost
  - Optimal path

Iterative Deepening Search (IDS) — Searching a Lost File
Question: 4
Files in a computer system are organized in a hierarchical directory tree.

- Represent directories as nodes in a tree.
- Implement **Iterative Deepening Search (IDS)** to locate a specific file name.
- Show how depth increases until the file is found.

Implement BFS, DFS using Python Classes
Question: 5
Design a **Graph class** in Python that includes:

- Methods to add nodes and edges
- BFS traversal method
- DFS traversal method

Use appropriate data structures:

- Queue for BFS
- Stack for DFS

Test the class on a sample graph and print traversal orders.

File Handling + Search Algorithm
Question: 6
A graph describing airline routes is stored in a text file:
Write a Python program to:

- Read the graph data from the file.
- Build the graph automatically.
- Implement **UCS** to find the cheapest route between two cities.
- Display the path and total cost.