University of Sunderland

CET 313 Artificial Intelligence

Workshop 3

Informed Search

Aims of the workshop

In the lecture, we introduce the concept of solving problem using informed search. Informed search is a type of search algorithm in artificial intelligence that uses additional information or heuristics to make more accurate decisions about which paths to explore first. These heuristics provide estimates of how close a given state is to the goal, guiding the search toward more promising solutions. Informed search is particularly useful in solving complex problems efficiently, as it can significantly reduce the search space and improve the speed of finding solutions.

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

Greedy Search: Greedy search is a type of informed search algorithm that makes the best decision in the current state without considering the complete problem. It evaluates each option based on the information available at the time, aiming to reach the goal state as quickly as possible. Greedy search chooses the path that appears to be the best at every step, based on a heuristic evaluation function. However, this approach does not guarantee the optimal solution, and it can sometimes get stuck in local optima.

A* search: is a widely-used informed search algorithm that combines the advantages of both Dijkstra's algorithm and Greedy Best-First Search. It uses both the cost to reach the current state (g-value) and the estimated cost to reach the goal from the current state (h-value). A* evaluates and selects the path that minimises the sum of these two costs, making it more efficient and effective compared to pure Greedy search. A* guarantees finding the shortest path from the initial state to the goal state, given an admissible heuristic.

Genetic Algorithm (GA): Genetic Algorithm is an optimisation algorithm inspired by the process of natural selection and genetics. It is commonly used to find approximate solutions to optimization and search problems. GA operates with a population of potential solutions (individuals), evolving these solutions over generations through processes such as mutation, crossover, and selection, similar to how organisms evolve in nature. It aims to improve the population's fitness by favouring individuals with better traits, eventually leading to a population that ideally represents the best solution to the given problem.

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

For Informed search, I recommend reading the following interactive blog:
https://www.redblobgames.com/pathfinding/a-star/introduction.html

For the Genetic Algorithm
Simple Genetic Algorithm From Scratch in Python - MachineLearningMastery.com

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.

Part 1: Informed Search Algorithm

Exercise 1: We will start with a simple problem that is the coin change problem. The goal of this problem is to determine the minimum number of coins needed to make a particular amount of change. The problem can be stated as follows: given a set of coin denominations and a target amount of change, the objective is to find the minimum number of coins required to make up that amount. Each coin has a specific value, and the goal is to minimise the total number of coins used while achieving the desired sum. The coin change problem is a well-known problem in the field of computer science and has various real-world applications, such as in vending machines, currency systems, and financial transactions.

Let us start with an example, say we have the following coin values: 1p, 2p, 5p, 10p, 20p, 50p, 100p, 200p. What is the minimum number of coins to make a 93p.

Take a few minutes to calculate the number. Write your solution in Jupyter notebook as a solution to exercise 1.

Exercise 2: **(Challenge 🧠)** Write a step-by-step instruction to find the min number of coins. Write your solution in Jupyter notebook as a solution to exercise 2.

Exercise 3: Use the following python script to solve the problem.

```
def coin_change(coins, amount):
    coins.sort(reverse=True)  # Sort the coins in descending order
    coin_count = 0  # Variable to keep track of the total number of coins used
    change = []  # List to store the coins used

    for coin in coins:
        while amount >= coin:
            amount -= coin
            coin_count += 1
            change.append(coin)

    if amount == 0:
        print(f" Minimum number of coins required: {coin_count}")
        print("Coins used:", change)
    else:
        print("Not possible to get the desired change with the coins.")
coins = [1, 2, 5, 10, 20, 50, 100, 200] # List of available coins
amount = 93 # Amount for which we need to find the minimum number of coins

coin_change(coins, amount)
```

Run the script to get the mini number of coins and compare it to your answer in exercise 1.

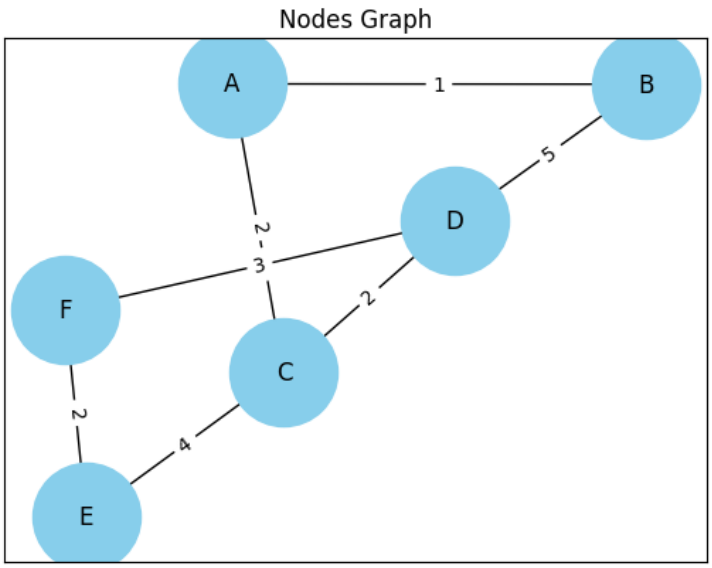
Exercise 4: Experiment with at least 3 different coins and 3 different sum amounts

Exercise 5: Trace the previous script and write a description of it.

Exercise 6: **(Challenge 🧠)** Is there anything you would change to enhance it?

Part 2: A* algorithm

Exercise 7: In this exercise we will use the A* to find the most optimal path in a graph.



To do so, we will need to instal networkx library, which can be installed using the following command:

```
pip install network
```

To read about it, you can refer to the library webpage:

NetworkX — NetworkX documentation

A. Initially we will start by defining a class for the nodes.

```
import heapq
import matplotlib.pyplot as plt
import networkx as nx

# Class representing a node in the graph
class Node:
    def __init__(self, name, heuristic_cost):
        self.name = name
        self.heuristic_cost = heuristic_cost
        self.adjacent = {}
        self.parent = None
        self.g_cost = float("inf")

    def add_neighbor(self, neighbor, cost):
        self.adjacent[neighbor] = cost

    def __lt__(self, other):
        return self.g_cost + self.heuristic_cost < other.g_cost + other.heuristic_cost
```

B. Define the A* algorithm.

```
# A* search algorithm
def astar_search(start, goal):
    open_list = []
    closed_set = set()

    start.g_cost = 0
    heapq.heappush(open_list, start)

    while open_list:
        current_node = heapq.heappop(open_list)

        if current_node == goal:
            path = []
            while current_node is not None:
                path.append(current_node.name)
                current_node = current_node.parent
            return path[::-1]

        closed_set.add(current_node)

        for neighbor, cost in current_node.adjacent.items():
            tentative_g_cost = current_node.g_cost + cost

            if neighbor in closed_set and tentative_g_cost >= neighbor.g_cost:
                continue

            if tentative_g_cost < neighbor.g_cost or neighbor not in open_list:
                neighbor.g_cost = tentative_g_cost
                neighbor.parent = current_node
                if neighbor not in open_list:
                    heapq.heappush(open_list, neighbor)
    return None
```

C. Then, we define our problem parameters:

```
# Creating nodes
A = Node("A", 5)
B = Node("B", 4)
C = Node("C", 3)
D = Node("D", 2)
E = Node("E", 1)
F = Node("F", 0)

# Adding neighbors and their costs
A.add_neighbor(B, 1)
A.add_neighbor(C, 2)
B.add_neighbor(D, 5)
C.add_neighbor(D, 2)
C.add_neighbor(E, 4)
D.add_neighbor(F, 3)
E.add_neighbor(F, 2)

# Visualization without the solution
G = nx.Graph()
edges = [(A.name, B.name, {'weight': 1}), (A.name, C.name, {'weight': 2}),
         (B.name, D.name, {'weight': 5}), (C.name, D.name, {'weight': 2}),
         (C.name, E.name, {'weight': 4}), (D.name, F.name, {'weight': 3}),
         (E.name, F.name, {'weight': 2})]
G.add_edges_from(edges)

pos = nx.spring_layout(G)
nx.draw_networkx(G, pos, with_labels=True, node_size=3000,
node_color='skyblue')
labels = nx.get_edge_attributes(G, 'weight')
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)

plt.title('Nodes Graph')
plt.show()
```

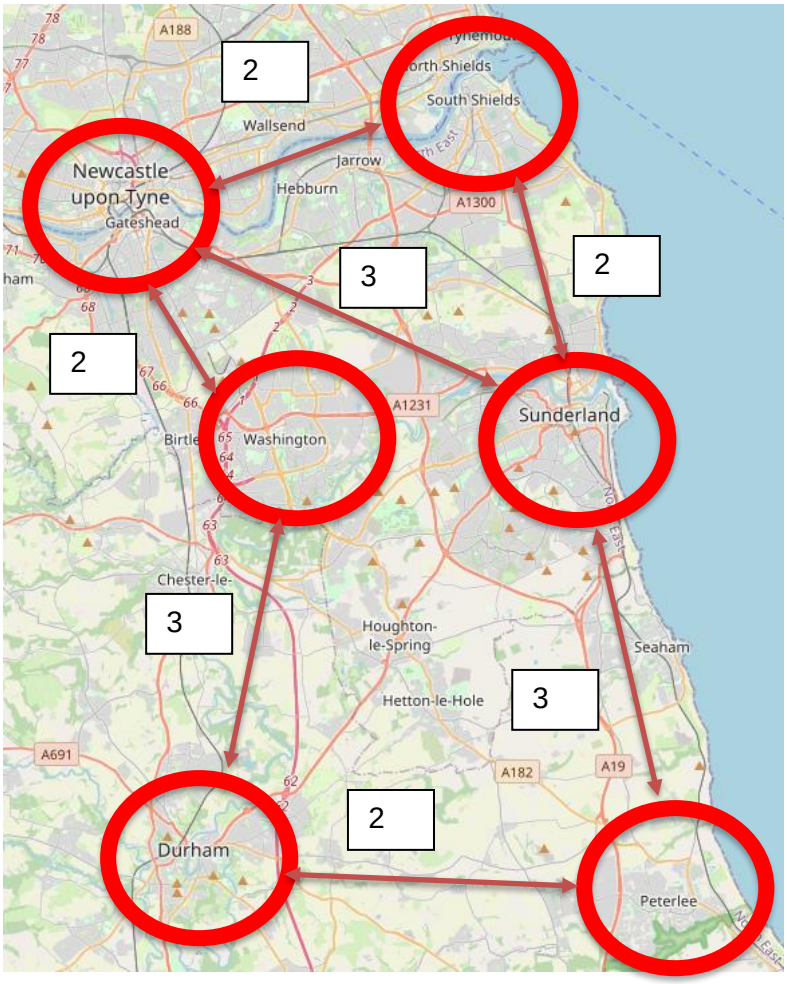
D. Now we have defined all the necessary parts we can run the A* algorithm and get the results.

```
# Running the A* search algorithm
path = astar_search(A, F)
print("A* path:", path)
```

E. Visualise the solution.

```
# Visualisation with A* path
plt.title('Graph with A* path')
nx.draw_networkx(G, pos, with_labels=True, node_size=3000,
node_color='skyblue')
nx.draw_networkx_edges(G, pos, edgelist=[(path[i], path[i + 1]) for i in
range(len(path) - 1)], edge_color='r', width=2)
nx.draw_networkx_edge_labels(G, pos, edge_labels=labels)
plt.show()
```

Exercise 8: **(Challenge 🧠)** Update the script to emulate the following graph, and let the starting point be in Sunderland and the end point be Washington.



Exercise 9: Use a maps website to get the actual distance between cities and update your script. Find the best path to get from Sunderland to Washington.

Exercise 10: Experiment with 3 different starting and ending points.

Part 3: Genetic Algorithm

Exercise 11: In this exercise we will use Genetic Algorithm to find a sentence in a vector game. Will start with a binary exercise to guess target value code **001010**

Here is a breakdown of the script:

1. **Initialization and Configuration**:
   - **POPULATION_SIZE**: Defines the number of individuals in each generation.
   - **GENES**: Represents the valid genes, in this case, it's '0' and '1'.
   - **TARGET**: Represents the target string that the algorithm is trying to generate.
2. **Individual Class**:
   - **mutated_genes**: Method to create a random gene for mutation from the global **GENES**.
   - **create_gnome**: Method to create a chromosome (a string of genes) of the same length as the **TARGET** by using the **mutated_genes** method.
   - **mate**: Method to perform mating and produce new offspring using crossover and mutation.
   - **cal_fitness**: Method to calculate the fitness score, which is the number of characters in the string that differ from the **TARGET** string.
3. **Main Function**:
   - Initialises the generation counter and a Boolean variable **found** to track whether the target string has been found.
   - Creates an initial population of individuals, each with their own chromosome.
   - Continuously evolve the population until the target string is found. It does so by selecting the fittest individuals for the next generation and allowing them to mate to produce offspring, while also applying elitism to ensure that the best-performing individuals are preserved.
   - Displays the generation number, the current string, and the fitness score of the fittest individual in each generation.
4. **Running the Algorithm**:
   - The script initiates the main function if the script is run directly, not imported as a module.

```
import random
# Number of individuals in each generation
POPULATION_SIZE = 4
# Valid genes
GENES = '''01'''
# Target string to be generated
TARGET = "001010"
```

```
class Individual(object):
    #Class representing individual in population
    def __init__(self, chromosome):
        self.chromosome = chromosome
        self.fitness = self.cal_fitness()
    @classmethod
    def mutated_genes(self):
        #create random genes for mutation
        global GENES
        gene = random.choice(GENES)
        return gene
    @classmethod
    def create_gnome(self):
        #create chromosome or string of genes
        global TARGET
        gnome_len = len(TARGET)
        return [self.mutated_genes() for _ in range(gnome_len)]
    def mate(self, par2):
        #Perform mating and produce new offspring
        # chromosome for offspring
        child_chromosome = []
        for gp1, gp2 in zip(self.chromosome, par2.chromosome):
            # random probability
            prob = random.random()
            # if prob is less than 0.45, insert gene from parent 1
            if prob < 0.45:
                child_chromosome.append(gp1)
            # if prob is between 0.45 and 0.90, insert gene from parent 2
            elif prob < 0.90:
                child_chromosome.append(gp2)
            # otherwise insert random gene(mutate),
            # for maintaining diversity
            else:
                child_chromosome.append(self.mutated_genes())
        # create new Individual(offspring) using generated chromosome for offspring
        return Individual(child_chromosome)

    def cal_fitness(self):
        ''' Calculate fitness score, it is the number of characters in string
which differ from target string. '''
        global TARGET
        fitness = 0
        for gs, gt in zip(self.chromosome, TARGET):
            if gs != gt: fitness+= 1
        return fitness
```

```
# Driver code
def main():
    global POPULATION_SIZE
    #current generation
    generation = 1
    found = False
    population = []
    # create initial population
    for _ in range(POPULATION_SIZE):
                gnome = Individual.create_gnome()
                population.append(Individual(gnome))
    while not found:
        # sort the population in increasing order of fitness score
        population = sorted(population, key = lambda x:x.fitness)
        # if the individual having lowest fitness score ie.
        # 0 then we know that we have reached to the target
        # and break the loop
        if population[0].fitness <= 0:
            found = True
            break
        # Otherwise generate new offsprings for new generation
        new_generation = []
        # Perform Elitism, that mean 10% of fittest population
        # goes to the next generation
        s = int((10*POPULATION_SIZE)/100)
        new_generation.extend(population[:s])
        # From 50% of fittest population, Individuals
        # will mate to produce offspring
        s = int((90*POPULATION_SIZE)/100)
        for _ in range(s):
            parent1 = random.choice(population[:50])
            parent2 = random.choice(population[:50])
            child = parent1.mate(parent2)
            new_generation.append(child)

        population = new_generation

        print("Generation: {}\tString: {}\tFitness: {}".format(generation,
            "".join(population[0].chromosome),
            population[0].fitness))
        generation += 1
    print("Generation: {}\tString: {}\tFitness: {}".format(generation,
        "".join(population[0].chromosome),
        population[0].fitness))
if __name__ == '__main__':
    main()
```

Exercise 12: Change the problem to have all the alphabets and update the target value to be ‘Welcome to AI’

Exercise 13: Update the script to have it so that it would measure the duration of finding the solution.

Exercise 14: **(Challenge 🧠)** Change the target to be 5 words and check how long it would take.

Exercise 15: Change the alphabet to have numbers and letters and repeat the previous exercise.

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

END OF EXERCISES