AI ECM2423 - Coursework exercise

690019495

March 24th, 2021

1 Q1

1.1

The puzzle consists of an area divided into a grid: 3x3 with one square empty. Thus, there are eight tiles in the puzzle. A tile that is next to the empty grid square can be moved into the empty space, leaving its previous position empty in turn. Tiles are numbered, 1 to 8 so that each tile can be uniquely identified.

This can be seen as a search problem, as each solution can have the optimal number of tiles to move. Therefore, each tile move will give a better or worse position. Starting from an initial node, the problem can be approached by aiming to find a path to the given solved puzzle goal node, where the path of nodes having the smallest cost (least moved tiles).

The best node parent will have the possible moves searched as children, which will be in turn be searched. The new parent node will be the puzzle with the best search value, and will be repeated until the optimal search path is found.

1.2.1

A * algorithm is a searching algorithm that searches for the shortest path between an initial and a final state. The A* algorithm has 3 parameters:

g : the cost of moving from the initial cell to the current cell. The sum of all the cells that have been visited since leaving the first cell.

h : The heuristic value, it is the estimated cost of moving from the current cell to the final cell. The actual cost will be calculated when the final cell is reached, ensuring h is never an over estimation of the cost. An algorithm is said to be admissible if it never overestimates the cost of reaching the goal.

f : The sum of g and h. Therefore f(n) = g(n) + h(n) where n is a node. Admissible if: $0 \leq h(N) \leq h*(N)$

h(n) is calculated using the heuristic function. With a non-admissible heuristic, the A* algorithm could overlook the optimal solution to a search problem due to an overestimation in f(n).

1.2.2

An admissible heuristic never overestimates the cost of reaching the goal. Using an admissible heuristic will always result in an optimal solution.

1: Sum of Manhattan distances of the tiles from their goal positions. Each tile will have a set of moves to its desired location Since you can only move the tiles 1 at a time and in only one of 4 directions, the optimal scenario for each block is that it has a clear, unobstructed path to its goal state.

Definition: The distance between two points measured along axes at right angles. In a plane with p1 at (x1, y1) and p2 at (x2, y2), $\|x1 - x2\| + \|y1 - y2\|$.

2: The Hamming distance (Number of misplaced tiles). This is an admissible heuristic, as every tile that is out of position must be moved at least once to its destination goal, and therefore never an overestimation.

Definition: The number of points measured not in the goal state. Where n in set X is not equal to the state of n in Y.

I selected these functions because for a given problem with a fixed value of the dimension, it is be preferable to use the Manhattan Distance as it is the most preferable for two dimensional applications. Also, the hamming distance will always be known to move a tile if it is out of the goal position. Therefore, these function can be easily implemented for the given task.

1.2.3

Code = 1.2.3.py

$$\text{Goal\_state} = \begin{bmatrix} 0 & 1 & 2 \\ 3 & 4 & 5 \\ 6 & 7 & 8 \end{bmatrix}$$

Configuration:

0 is the state of the empty tile, and all moves can be made to the non-diagonal elements of the matrix. (Up, Down, Left, Right) As long as these moves are valid.

Using default search:

$$\text{puzzle.set(”325687014”) (optimal in 16 moves)} = \begin{bmatrix} 3 & 2 & 5 \\ 6 & 8 & 7 \\ 0 & 1 & 4 \end{bmatrix}$$

are one of the many options that can be commented using # to select the default puzzle in the code.

Each run will produce the optimal node path in the terminal, which can be followed to the solution.

1.2.4

The Manhattan distance heuristic is far more effective in node generation and queue sorting. The f(n) value is more easy to calculate and the nodes generated have better depth quality when selecting the next node.

Comparing results given by A* when using the two different heuristic functions:

puzzle.set("325687014") (16 moves) = $\begin{bmatrix} 3 & 2 & 5 \\ 6 & 8 & 7 \\ 0 & 1 & 4 \end{bmatrix}$

Manhattan function results ::

In moves: 16 , Depth: 15, Nodes visited: 197, Total nodes generated: 326

Misplaced tiles results ::

In moves: 16, Depth: 15, Nodes visited: 622, Total nodes generated: 1007

The Manhattan heuristic function generated 1/3 as many nodes as the Misplaced heuristic. Furthermore visiting over 700 less nodes, resulting in a faster execution time.

puzzle.set("724506831") (26 moves) = $\begin{bmatrix} 7 & 2 & 4 \\ 5 & 0 & 6 \\ 8 & 3 & 1 \end{bmatrix}$

The Manhattan Distance ::

In moves: 26, Depth: 25, Nodes visited: 3488, Total nodes generated: 5564

Where the Misplaced heuristic was not even comparable, with node generation exceeding 20000. The Misplaced tile function is very inefficient, when comparing large node generations, the misplaced tile heuristic becomes worse the deeper the depth.

1.3

Code = 1.3_Astar.py

User can input any state and goal (that can be solved). User must enter a start state and a goal sate with the same corresponding numbers as the start state (or it cannot be solved)

Example:

Start state : 0 1 2 3 4 4 4 5 8

Goal state : 4 4 4 5 8 1 2 3 0

In moves: 26, Depth: 25, Nodes visited: 3227, Total nodes generated: 5009

Further discussions

Comparing the outcomes of randomly generated puzzles for Manhattan, Misplaced and Euclidean distance for the value of h(n). These values are averaged for similar puzzles with the same number of moves needed to complete.

```
def euclideanDistance():

search = (1,2,3,4,5,6,7,8,0)

goal = (0,1,2,3,4,5,6,7,8)

distance = math.sqrt(sum([(a - b) ** 2 for a, b in zip(search, goal)]))

return distance
```

puzzle.set("123405678") (14 moves)

Method	Nodes Explored	Total Nodes generated
Manhattan	131	215
Euclidean	448	732
Misplaced	480	796

puzzle.set("325687104")(15 moves)

Method	Nodes Explored	Total Nodes generated
Manhattan	145	240
Euclidean	352	571
Misplaced	518	846

puzzle.set("032568714") (18 moves)

Method	Nodes Explored	Total Nodes generated
Manhattan	317	518
Euclidean	1493	2370
Misplaced	2718	4352

Overall, Manhattan Distance is more efficient than both the other two methods and out performing the iterations of the puzzles. Euclidean has similar performance to misplaced at smaller puzzle sets, but for the larger

2 Q2

2.1

K-means algorithm is an iterative algorithm that tries to partition the dataset into K pre-defined distinct non-overlapping subgroups (clusters) where each data point belongs to only one group.

Goal: given centroids a(1).....a(k), assign each point to the best centroid

The way k-means algorithm works: Specify number (n) of clusters K. Initialize centroids by first shuffling the dataset and then randomly selecting K data points for the centroids without replacement. Keep iterating until there is no change to the centroids Compute the sum of the squared distance between data points and all centroids. Assign each data point to the closest cluster (centroid). Compute the centroids for the clusters by taking the average of the all data points that belong to each cluster.

This algorithm approach can be applied to the task of hand-written digit recognition as the size of the dataset is known, which allows the user to assign the 10 clusters to each number. The k-means algorithm can be trained to understand the numbers and identify from the cluster what the written number is.

2.2.1

Code == dataShow.py

2.2.2

I have not implemented my own version of the Kmeans algorithm.

2.2.3

Using the mask it is clear outline for the clusters and what digits fall into each cluster. The centroids are accurate according to the cluster density, however the clusters are more spread out for the digits 2 3 and 5 as these have the most similar attributes in the dataset. Overall the Kmeans clusters are fairly accurate with a maximum iteration number of 300.

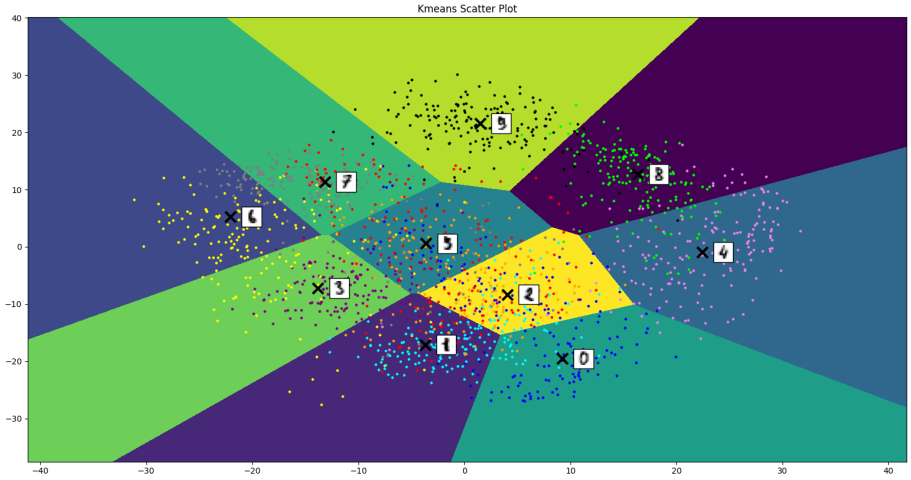


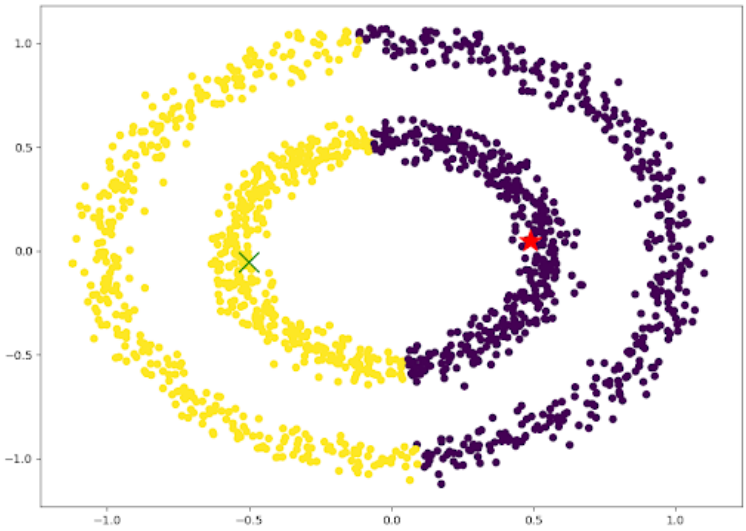
Figure 1: Kmeans using a mask to identify the fit. Centroid plots show the numbers associated. Mask citation: https://scikit-learn.org/stable/auto_examples/cluster/plot_kmeans_digits.html

Examples of the digits in the sklearn.dataset load_digits.

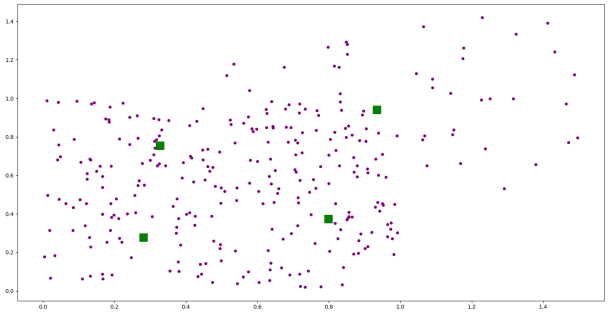
2.3

The algorithm is only applicable if the mean is defined. K-means algorithm is good in capturing the structure of data if the clusters have a spherical-like shape. It will always try to construct a nice spherical shape around the centroid. Additionally, the user needs to specify k.

Ex1 - if the clusters have a complicated geometric shapes, k-means does a poor job in clustering the data. As humans, we would cluster the individual rings as those clusters, while Kmeans assumes to split the two into the respective clusters.



Ex2 K-Means clustering algorithm is most sensitive to outliers as it uses the mean of cluster data points to find the cluster centre. If the data has many outliers, then k-means is more likely to fail. In this example, there is a cluster in the top right more spread out, and a cluster in the bottom right. The 4 assigned clusters are relatively inn-accurate where the clusters are believed to be based.



Green boxes show centroids