Comparison of Heuristics for A-star Search

# Hrishikesh V (PES1201700276)

# Laxman (PES120170\_\_\_)

# Ravendra Singh (PES120170070\_)

# What is A-Star Search

A \* search is a best first search (Informed Search) algorithm which is used mainly because it’s complete, optimal and efficient. It is applied for graph traversal and path search problems and has a space complexity of O(bd).

When applied to a shortest path problem, this algorithm aims to find the shortest path given the graph and the end points. At each step, it constructs the shortest path from the source along the direction of the destination, and increments the path one edge at a time until the destination is reached.

While this appears to be similar to greedy algorithms, it is in fact not the case. As opposed to A-star search, Greedy algorithms tend to get stuck in locally optimal solutions while the former avoids doing so with the help of a heuristic function.

For any given graph, the path that minimizes the function **f(x) = g(x) + h(x)** is considered to be the optimal solution.

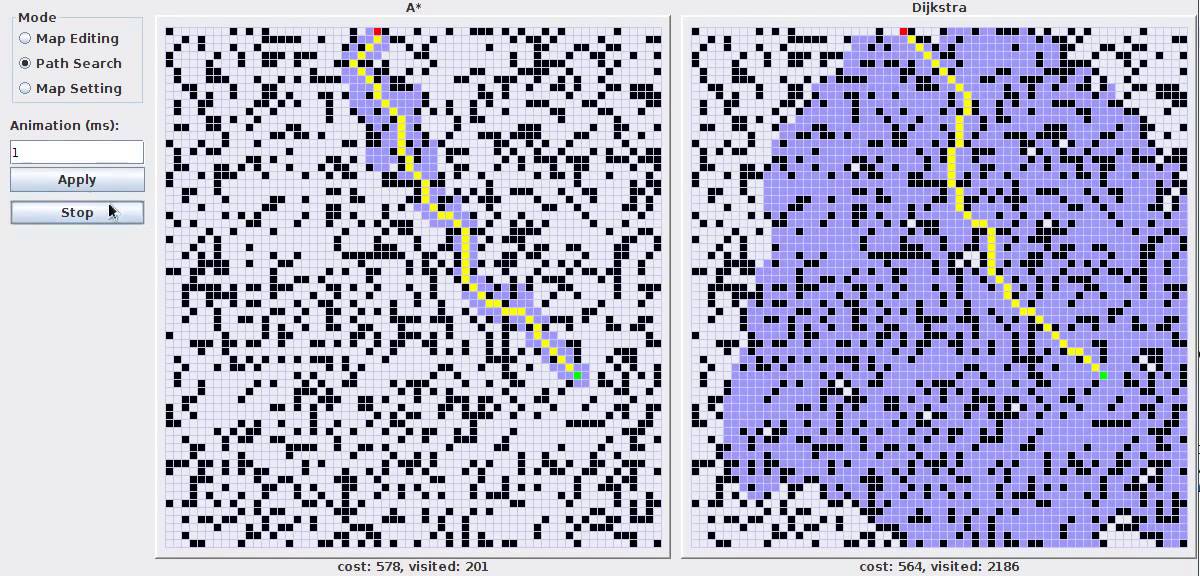
The assumption here is that the actual cost of traversing from the source node to an intermediate node is known before reaching the destination. This information, combined with the heuristic that determines the approximate cost of reaching the destination from the intermediate node, makes this algorithm reliable.

# A-star search Algorithm

1. Add the first node to the list
2. Find the path to the nearest neighbor of the first node.
3. If there is no path, ignore the node, otherwise, add it to the list
4. Use G(X) to determine the lowest cost to traverse to the nearest neighbor from the current node
5. Use a heuristic to move in the direction of the destination
6. Repeat until destination is reached.

# Dijkstra’s algorithm vs. A-star search

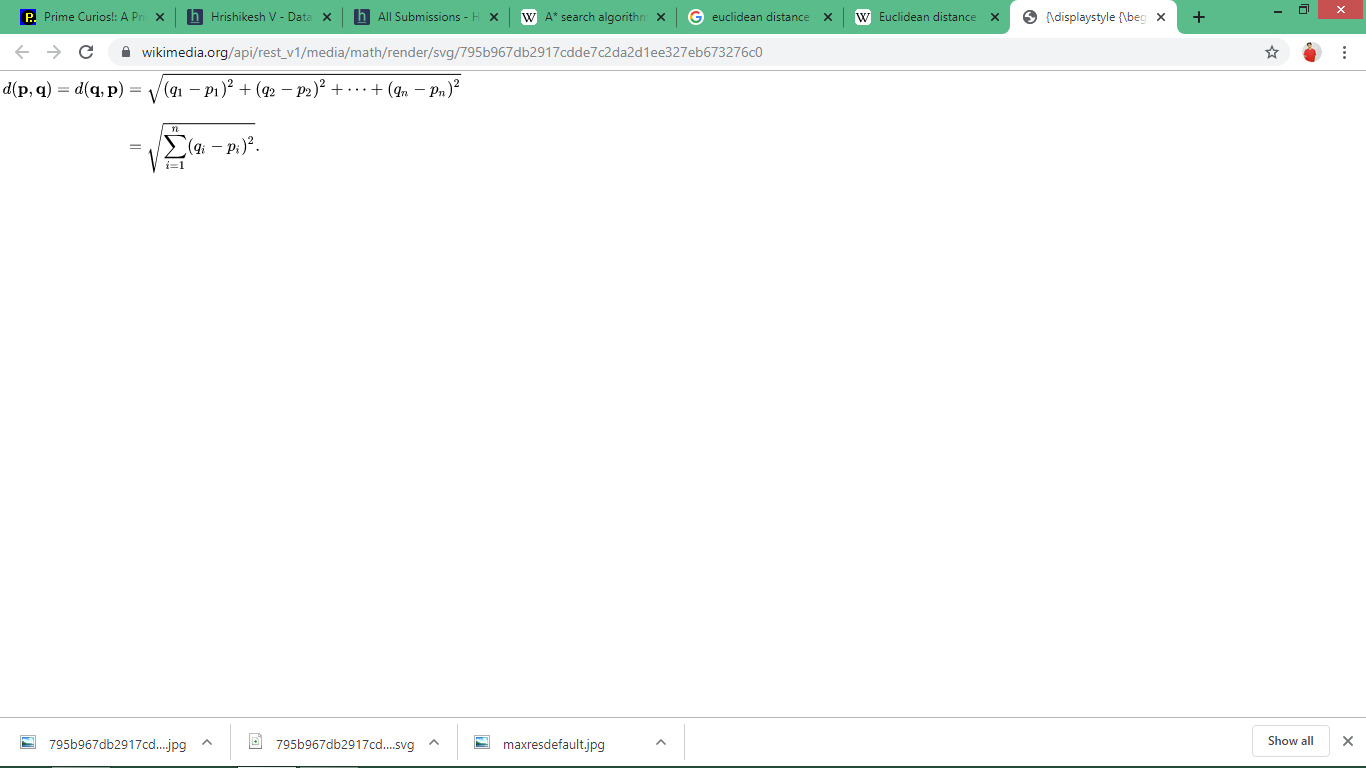
Dikstra’s algorithm is widely used in problems where the shortest path is calculated. This algorithm is also a uniform cost search algorithm not unlike A\* search and can be considered as a special case of A\* search where the heuristic function is 0 for all x.



# Different Heuristics that can be used for path finding problems

## 1 Euclidean Distance

This is also called the straight line distance between two points. It is the length of the line segment that connects two points in a n-dimensional space



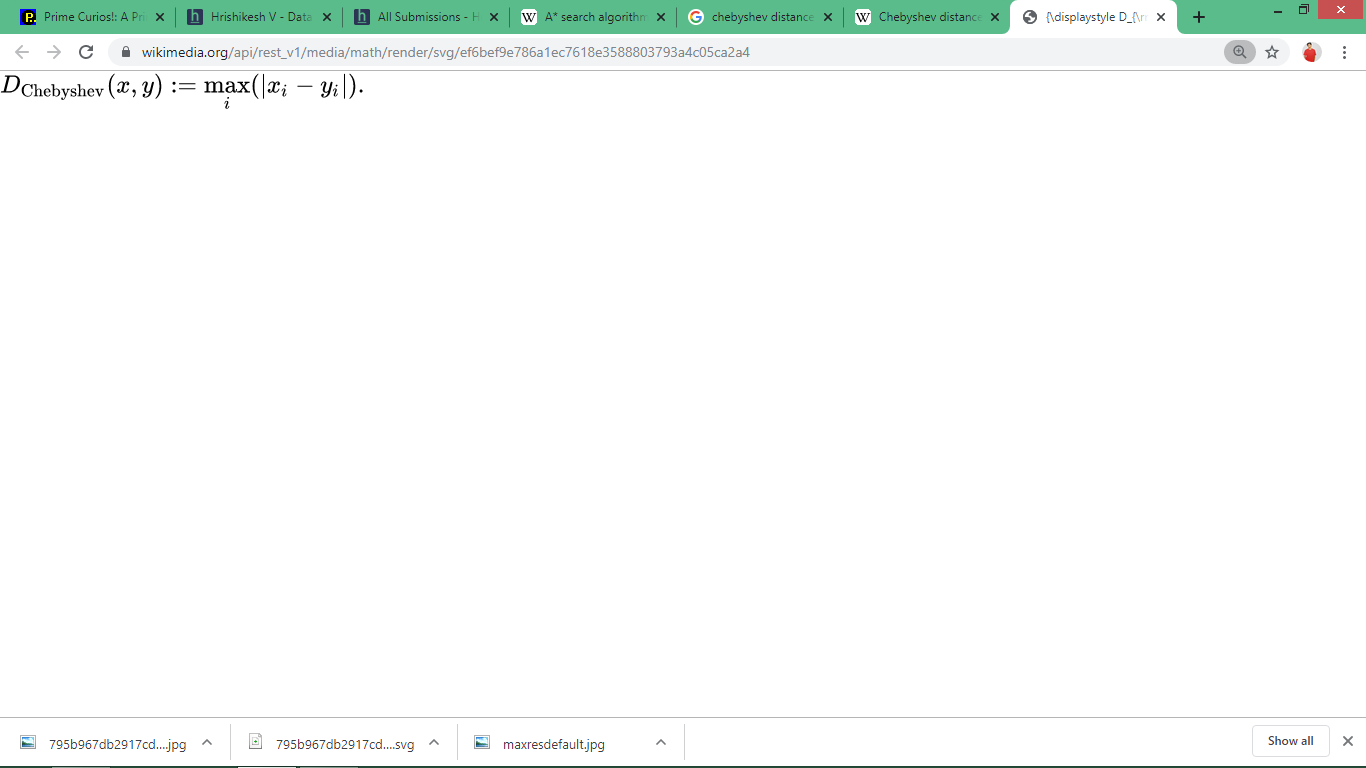
## 2 Manhattan Distance

Manhattan distance is the distance between two points in an n-dimensional space measured along axes at **right angles.**

D(p, q) = |x1 - x2| + |y1 – y2|

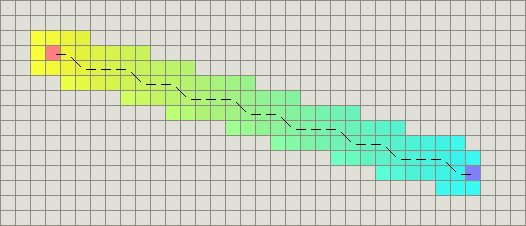
## 3 Chebyshev Distance

For any two vectors in a vector space, chebyshev longest distance between the two along any co-ordinate dimension. It is also called chessboard distance.



## 4 Octile Distance

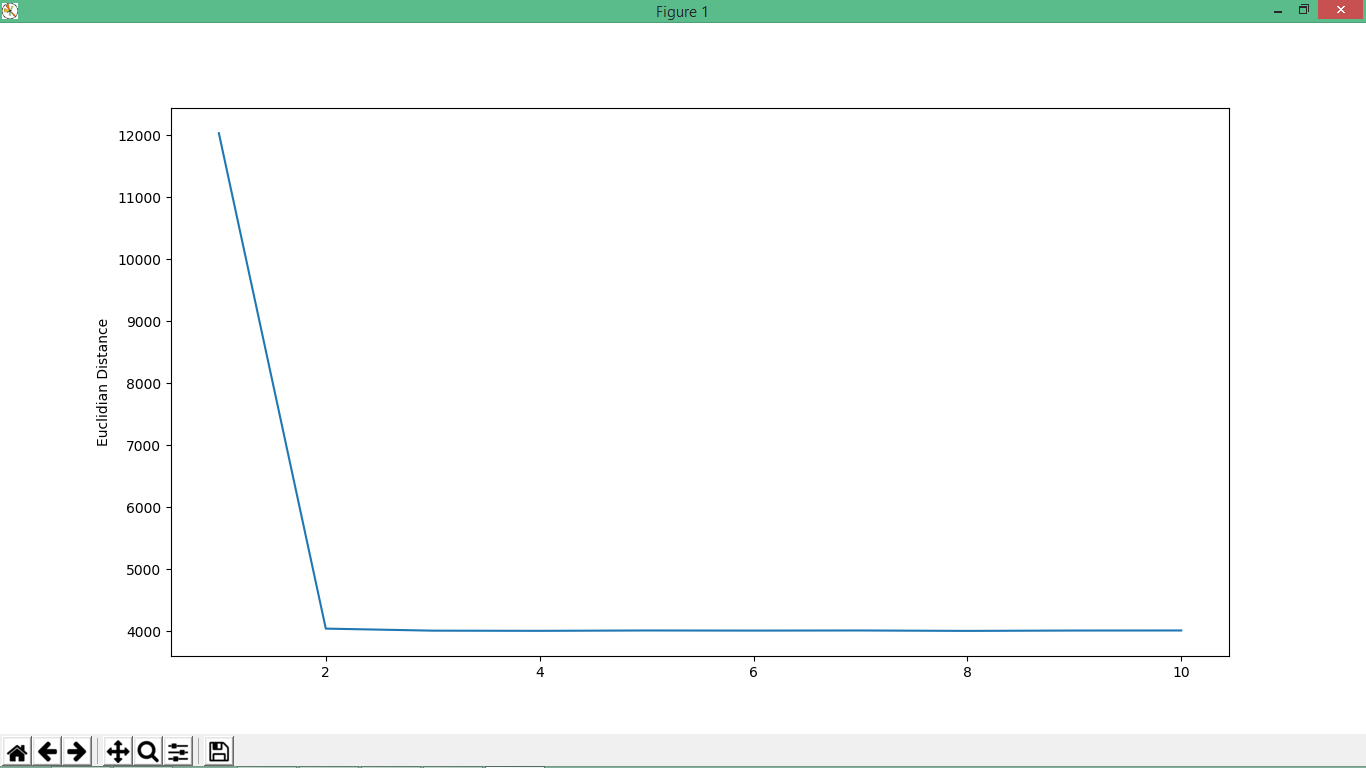
Octile distance is a special case of weighted Chebyshev distance. It calculates the distance along the diagonals.



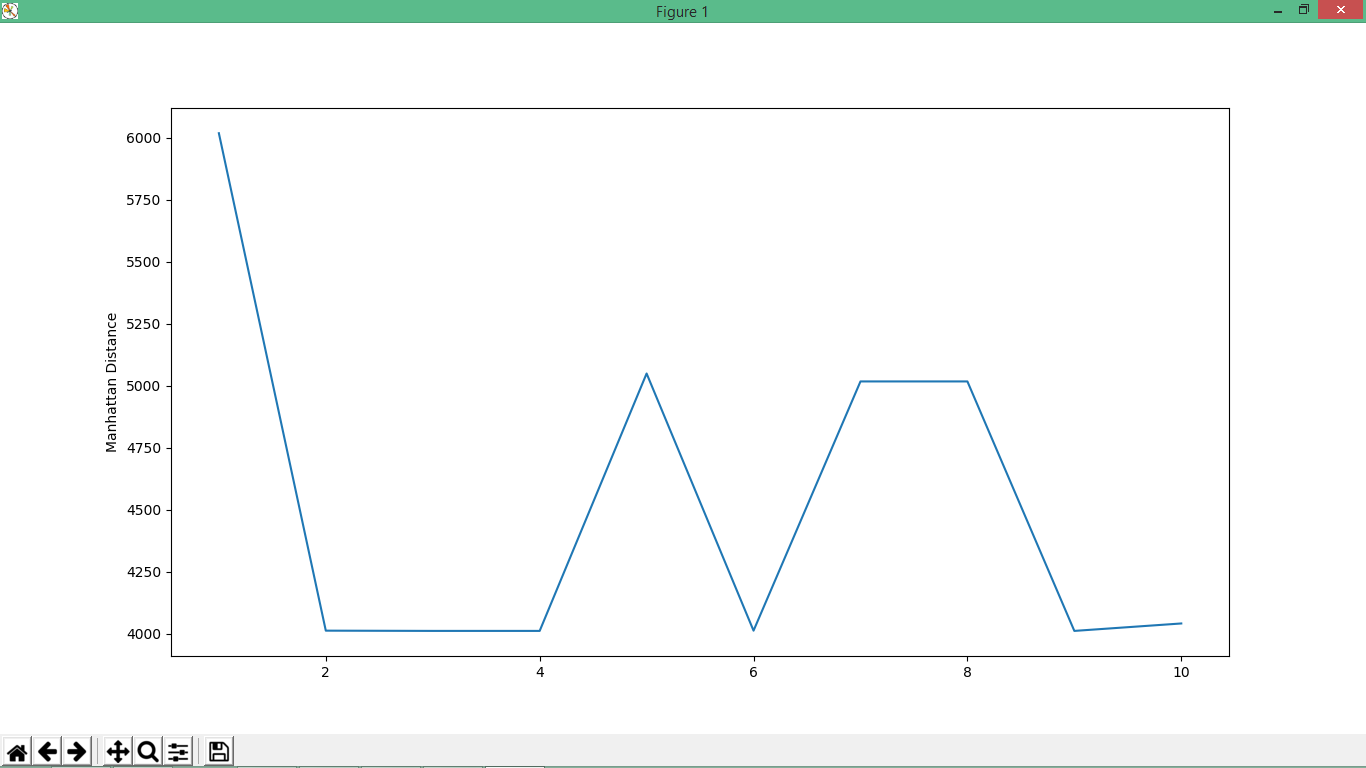
# Time complexity of Heuristics

Here are the graphs of time taken (In microseconds) by various heuristics over 10 trials on an arbitrary adjacency matrix.

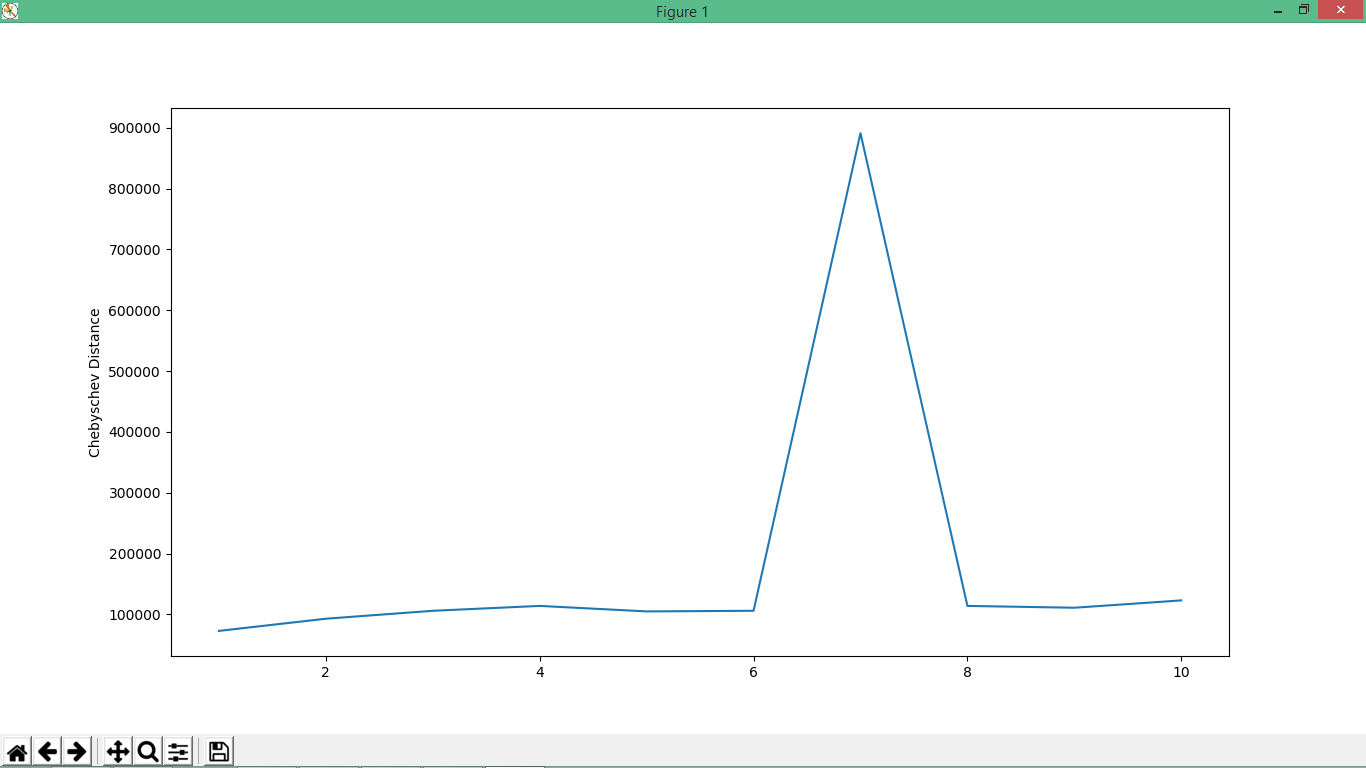
**Euclidean Distance**



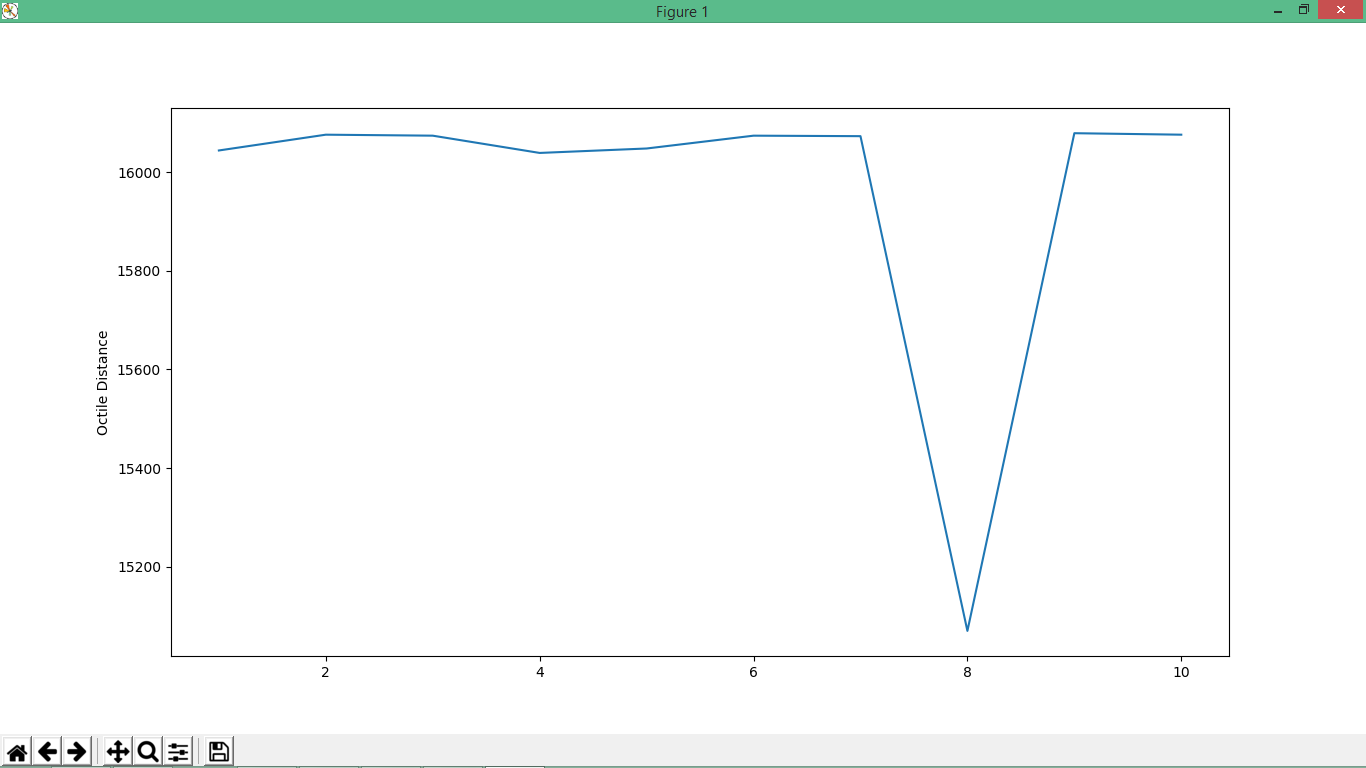
**Manhattan Distance**



**Chebyshev Distance**



**Octile Distance**



# Conclusion

While all of the above heuristic functions succeed in finding the optimal solution, their time complexities vary significantly.

Manhattan distance is found to be consistently faster than all the other heuristics while chebyshev distance is takes the longest to find the optimal path.

# Code (Python)

from datetime import datetime

import matplotlib.pyplot as plt

class Node():

def \_\_init\_\_(self, parent=None, position=None):

self.parent = parent

self.position = position

self.g = 0

self.h = 0

self.f = 0

def \_\_eq\_\_(self, other):

return self.position == other.position

def astar(maze, start, end, heuristic):

start\_node = Node(None, start)

start\_node.g = start\_node.h = start\_node.f = 0

end\_node = Node(None, end)

end\_node.g = end\_node.h = end\_node.f = 0

open\_list = []

closed\_list = []

open\_list.append(start\_node)

while len(open\_list) > 0:

current\_node = open\_list[0]

current\_index = 0

for index, item in enumerate(open\_list):

if item.f < current\_node.f:

current\_node = item

current\_index = index

open\_list.pop(current\_index)

closed\_list.append(current\_node)

if current\_node == end\_node:

path = []

current = current\_node

while current is not None:

path.append(current.position)

current = current.parent

return path[::-1] # Return reversed path

children = []

for new\_position in [(0, -1), (0, 1), (-1, 0), (1, 0), (-1, -1), (-1, 1), (1, -1), (1, 1)]: # Adjacent squares

node\_position = (current\_node.position[0] + new\_position[0], current\_node.position[1] + new\_position[1])

if node\_position[0] > (len(maze) - 1) or node\_position[0] < 0 or node\_position[1] > (len(maze[len(maze)-1]) -1) or node\_position[1] < 0:

continue

if maze[node\_position[0]][node\_position[1]] != 0:

continue

new\_node = Node(current\_node, node\_position)

children.append(new\_node)

for child in children:

for closed\_child in closed\_list:

if child == closed\_child:

continue

child.g = current\_node.g + 1

if heuristic == 0:

#Euclidian Distance

child.h = ((child.position[0] - end\_node.position[0]) \*\* 2) + ((child.position[1] - end\_node.position[1]) \*\* 2)\*\*0.5

elif heuristic == 1:

#Manhattan Distance

child.h = abs(child.position[0] - end\_node.position[0]) + abs(child.position[1] - end\_node.position[1])

elif heuristic == 2:

#Chebyschev Distance

dx = abs(child.position[0] - end\_node.position[0])

dy = abs(child.position[1] - end\_node.position[1])

D = 1

D2 = 1

child.h = D \*(dx + dy) + (D2 - 2\*D)\*min(dx, dy)

else:

#Octile Distance

dx = abs(child.position[0] - end\_node.position[0])

dy = abs(child.position[1] - end\_node.position[1])

D = 1

D2 = 2\*\*0.5

child.h = D \*(dx + dy) + (D2 - 2\*D)\*min(dx, dy)

child.f = child.g + child.h

for open\_node in open\_list:

if child == open\_node and child.g > open\_node.g:

continue

open\_list.append(child)

def main():

maze = [[0, 0, 1, 0, 1, 0, 0, 1, 0, 0],

[0, 0, 1, 1, 1, 1, 0, 0, 0, 1],

[0, 0, 1, 0, 1, 0, 0, 1, 0, 0],

[1, 0, 1, 0, 1, 1, 1, 1, 0, 0],

[0, 1, 0, 1, 0, 1, 0, 0, 1, 1],

[1, 0, 0, 0, 1, 0, 1, 1, 0, 1],

[0, 0, 1, 0, 1, 1, 0, 0, 0, 0],

[0, 1, 0, 0, 1, 0, 0, 1, 1, 1],

[0, 1, 0, 1, 1, 1, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 1, 0, 0, 0]]

maze2 = [[0, 0, 1, 0, 1, 0, 0, 1, 0, 0,0, 0, 1, 0, 1, 0, 0, 1, 0, 0],

[0, 0, 1, 1, 1, 1, 0, 0, 1, 1,0, 1, 1, 1, 1, 1, 1, 1, 0, 1],

[0, 0, 1, 0, 1, 0, 1, 1, 1, 0,1, 1, 1, 1, 1, 1, 1, 1, 0, 0],

[1, 0, 1, 0, 1, 1, 1, 1, 0, 1,1, 0, 1, 0, 1, 1, 1, 1, 1, 0],

[0, 1, 0, 1, 0, 1, 0, 0, 1, 0,0, 1, 0, 1, 0, 1, 1, 1, 1, 1],

[1, 0, 0, 0, 1, 0, 1, 1, 0, 1,1, 0, 0, 0, 1, 1, 1, 1, 1, 1],

[0, 0, 1, 0, 1, 1, 0, 0, 0, 0,0, 0, 1, 0, 1, 1, 0, 0, 0, 0],

[0, 1, 0, 0, 1, 0, 0, 1, 1, 1,1, 0, 0, 0, 1, 0, 0, 1, 1, 1],

[0, 1, 0, 1, 1, 1, 0, 0, 0, 0,0, 1, 0, 1, 1, 1, 0, 0, 0, 0],

[0, 0, 0, 0, 0, 0, 1, 0, 0, 0,0, 0, 0, 0, 0, 0, 1, 0, 0, 0]]

start = (0, 0)

end = (4, 14)

eucdist = [];

mandist = [];

chebdist = [];

octdist = [];

i = 0

y = [1,2,3,4,5,6,7,8,9,10]

while(i<10):

start\_t = datetime.now().microsecond

path = astar(maze2, start, end,0)

end\_t = datetime.now().microsecond

eucdist.append(abs(end\_t-start\_t))

print("Time taken when heuristic is Euclidian distance is :" + str(end\_t-start\_t))

print(path)

print(len(path))

start\_t = datetime.now().microsecond

path = astar(maze2, start, end, 1)

end\_t = datetime.now().microsecond

mandist.append(abs(end\_t-start\_t))

print("Time taken when heuristic is Manhattan distance is :" + str(end\_t-start\_t))

print(path)

print(len(path))

start\_t = datetime.now().microsecond

path = astar(maze2, start, end, 2)

end\_t = datetime.now().microsecond

chebdist.append(abs(end\_t-start\_t))

print("Time taken when heuristic is Chebyschev distance is :" + str(end\_t-start\_t))

print(path)

print(len(path))

start\_t = datetime.now().microsecond

path = astar(maze2, start, end, 3)

end\_t = datetime.now().microsecond

octdist.append(abs(end\_t-start\_t))

print("Time taken when heuristic is Octile distance is :" + str(end\_t-start\_t))

print(path)

print(len(path))

i = i + 1

print("Euclid distance")

plt.plot(y, eucdist)

plt.ylabel('Euclidian Distance')

plt.show()

print(eucdist)

print("Manhattan Distance")

plt.plot(y, mandist)

plt.ylabel('Manhattan Distance')

plt.show()

print(mandist)

print("Chebyschev Distance")

plt.plot(y, chebdist)

plt.ylabel('Chebyschev Distance')

plt.show()

print(chebdist)

print("Octile Distance")

plt.plot(y, octdist)

plt.ylabel('Octile Distance')

plt.show()

print(octdist)

if \_\_name\_\_ == '\_\_main\_\_':

main()