Improved A* Algorithm via the use of K-means Clustering

Submission Date · July 2019

Author **:** Joseph Assaker - 53190
~ 4th Year Computer Science Student (M1)

Supervisor **:** Dr. Joseph Constantine

Improved A* Algorithm via the use of K-means Clustering

Abstract *- A* (pronounced "A star") is a computer algorithm that is widely used in pathfinding and graph traversal, which is the process of finding a (optimal) path from point A to point B, while of course traversing other points n_i in between. These points are referred to as "Nodes". In the present work, we aim at optimizing the standard A* algorithm by firstly proposing a better performing version of that algorithm that is possible due to an exact heuristic function, and secondly by optimizing the output path of the algorithm via the use of the K-means algorithm, which we will briefly presented in this paper as well. Accordingly, we will be conveying our explanations and demonstrating the algorithms' results and performances via the use of a real world map as a general example.*

Keywords: ***A star, Graph Theory, Pathfinding, K-means, Traffic, SUMO.***

I. INTRODUCTION

Graph traversal and optimal pathfinding given a cost function to optimize (e.g., distance, time, energy, etc...) has been a subject of study and interest for about 300 years now, dating back to Leonhard Euler's "Seven Bridges of Königsberg" published in 1736 [OX001]. That interest is in fact attributed due to the immense field of applications of such studies, ranging from applications in games [IJ001] to problems such as the problem of parsing using stochastic grammars in NPL [BK001]. Both previously mentioned applications utilise the A* pathfinding algorithm, a weight-based best-first algorithm which emerged as a provably optimal solution for pathfinding, outperforming older pathfinding algorithms such as BFS, DFS and Hill climbing; It can also be viewed as an extension to Dijkstra's Algorithm [WK002].

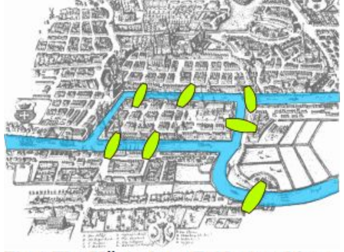

Fig.1 The Königsberg Bridge problem

A* was created as part of the Shakey project, which was the first general-purpose mobile robot to be able to reason about its own actions (1966 - 1972) [WK003].

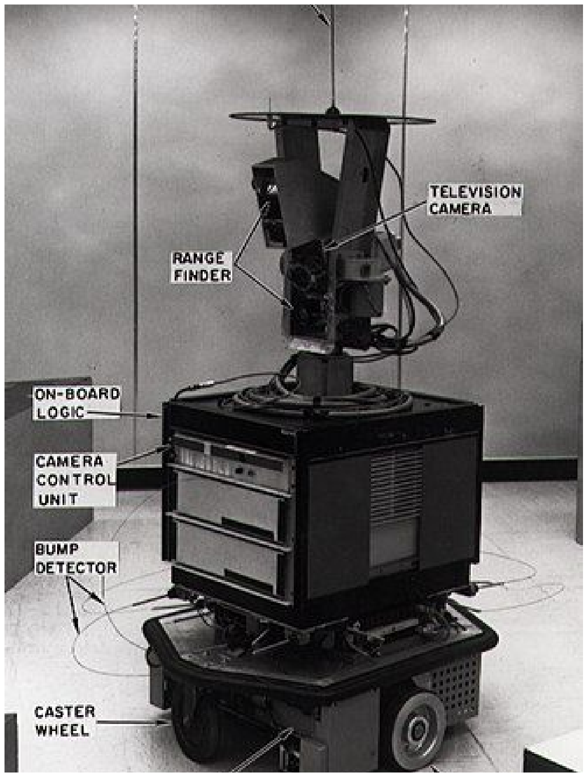

Fig.2 Shakey the Robot's path planning - "Which had the aim of building a mobile robot that could plan its own actions"

Although many variations and alternatives of the A* algorithm have been recently researched [RG001] [SS001], in this paper we will be using the original A* algorithm as our starting point and the end goal to be optimized.

The A* algorithm is an informed search algorithm which is formulated in terms of weighted graphs. The goal is to find the optimal path from point A to point B, and that is achieved by maintaining a tree of paths originating at the origin node (A) and extending those paths, one edge at a time, until its termination criterion is satisfied (e.g., current node is the destination B).
First, let's define the cost function f(n) = g(n) + h(n), where n is the next hop node, g(n) is the cost from the current node to n, and h(n) the estimated lowest cost for a given path from n to the destination B (h(n) is called the heuristic function). In a more general view, let $\pi = n_0, n_1, n_2, \ldots, n_p$ be a path whose last extended node is n_p (and is of course originated at $A = n_0$). The cost function for this path is $f(\pi) = g(\pi) + h(\pi)$, where $g(\pi)$ is the exact cost to go from node A (n_0) to node n_p, following the path $\pi = n_0, n_1, n_2, \ldots, n_p$, and $h(\pi)$ is the heuristic function of the node n_p: $h(n_p)$; That is the expected lowest cost for a given path from n_p to B (i.e., following a path $\pi = n_p, n_{p+1}, \ldots$, B).
At each iteration the algorithm decides which path (from the maintained tree of paths in memory) to extend next, and the path with the lowest cost $f(\pi_i)$ associated with it is elected to be extended (i.e., we store the currently explored-extendable paths available in memory in a sorted ascending list based on each path's cost function, and we always pick the first element in this list as our next path to be extended). Next we decide to which node (i.e., following which edge) we extend the elected path, and that is determined in such a way to optimize (minimize) the cost function: f(n) = g(n) + h(n) at each iteration.

In order to convey the ideas and show results and outputs in this paper, we will be using a real world map as our core example. This will be the map of an area of Beirut-Lebanon, exported from Open Street Map's official website [OM001], and then via the use of SUMO (Simulation of Urban MObility) [SM001] we were able to convert the initial map.osm file to a map.net.xml file that can be easily parsed by Python's built in libraries, as we will be using Python as our coding language for this paper. Furthermore, our initial goal will be to optimize the performance of A* while outputting the shortest (in distance) possible path between two points, and our secondary goal would be to further optimize that algorithm in order to find the quickest (in time) possible path between two points on the map.

The remainder of the paper is organized as follows. Section 2 discusses an initial performance optimization of the A* algorithm possible due to a perfect heuristic function elaborated in the beginning of the same section. Section 3 introduces the idea of traffic in the map and gives an overview of the K-means algorithm, then proceeds by implementing and utilizing the output of that algorithm in such a way to optimize the output performance of the A* algorithm. Section 4 elaborates some results and discusses some failed attempts to optimize the traffic distribution. Section 5 concludes with a brief description of future works.

II. Optimizing A* Algorithm Via a Perfect Heuristic Function

As defined in the introduction section, the cost function f(x) is the sum of two other functions, g(x) and h(x). As g(x) is always static, as it represents the edges' weight, therefore f(x) correctness and efficiency is entirely dependent on h(x), the heuristic function, which we have complete control over. A better heuristic function yields to a more effective cost function which in turn leads to a better performing A* algorithm. Fig.3 shows the difference in performance (i.e., how many tested paths before finding the solution; The shortest path) between two A* star algorithms, with the image on the left showing an average heuristic function, and the image on the right showing the use of a much more efficient heuristic function (to note that this is an extreme case, that slightly hinders the output performance).

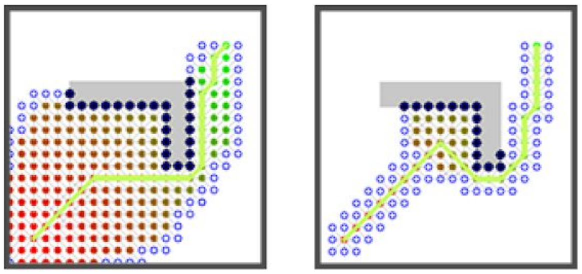

Fig.3 Heuristic function impact on the performance of A* algorithm - "All filled circles are visited nodes with greener colors being closer to the destination, empty circles are possibly extendable nodes, and the grey area representing an obstacle"

Having that in mind, we assume that we have a perfect heuristic function, which means that h(n) is no longer an "estimate" but rather the exact cost for the shortest path between n and the destination. Further observation reveals that with such a heuristic function, $f(\pi_i)$ for $i \in [0..p]$, where $\pi_i = n_0, \ldots, n_i$ and $n_0 = A$ and $n_p = B$, have the same value for every π_i.

E.g., let A(origin) → C → B(destination) be a weighted graph with A→ C of weight 5 and C → B of weight 2. An exact heuristic function should attribute for each node the exact lowest cost path to the destination B, thus h(A) = 7, h(C) = 2 and h(B) = 0. Furthermore:

$f(\pi_0 = [A])$ = 0(distance to go from A to A) + h(A) = 7.

$f(\pi_1 = [A, C])$ = 5 (distance to go from A to C) + h(C) = 7.

$f(\pi_2 = [A, C, B])$ = 5 + 2 (distance to go from A to B) + h(B) = 7.

This observation allows us to formulate the following fundamental axiom for our optimized A* algorithm: When deciding to which node to extend next, we only pick the node n that has g(n) + h(n) = h(current node), because that is the exact node from which the shortest path from the current node to the destination is formulated. That axiom allows us to completely omit the tree of currently expandable paths in memory, and thus only have the optimal path stored in memory, which is expanded exact node by exact node at each iteration until the destination is reached.

However, the price to pay in that case, would be of course the distribution of the exact heuristic function for all nodes of the graph. That is achieved by starting from the destination with a heuristic cost of 0, and expanding from the destination to all nodes, updating each node n with the exact cost from n to destination. If it is the first time we visit a node n, we assign for h(n) the value of the heuristic function of the node m, node from which we reached n from, h(m) plus the edge weight from n to m. If it is not the first time we visit n, we attribute to h(n) = min(h(n), h(m) + weight from n to m). And we only visit or revisit node n's neighbours if h(n) was initialized or updated (i.e., if h(n) remained the same after h(n) = min(h(n), h(m) + weight(m→n)), that means that its neighbours are already updated by the lowest cost, and no need to revisit them from this node).

In order to implement the heuristic function distribution algorithm, we need to use an array of nodes to be visited (initiated with the destination node), and at each iteration: (1) pop the first node, (2) process it, (3) push its neighbours into the array if necessary, and (4) repeat till the array is empty. Having the choice to either utilize the BFS spirit (FIFO array) or the DFS spirit (LIFO array), further analysis reveal that the BFS spirit is much more suited for such a task. As if we use the DFS spirit, we will be updating a large amount of nodes successors of a node whose heuristic function has a high chance of not being final, so when we come back to this node, and eventually update its heuristic function, we will then need to revisit each successor node again and again. In contrast, the BFS spirit allows us to updates each "layer" at a time and be certain about its heuristic values, before moving to the successors of those nodes. In fact, Fig.4 confirms this analysis by comparing the average processor time needed to calculate the exact heuristic function on our example map having 2373 nodes, using a FIFO and a LIFO array.

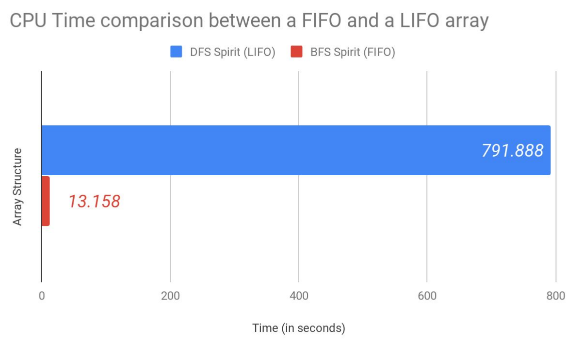

Fig.4 CPU Time comparison between a FIFO and a LIFO array - "This chart confirms the proposed analysis, and thus the BFS Spirit (using a FIFO array) will be used in our final model"

Having done this part, and having the exact heuristic function ready for a given destination B, we can now proceed by finding the shortest path (optimal in distance) from any point n on the graph to the destination B in a linear CPU time, based on the number p of nodes of the shortest path between the origin and the destination. In practice, the average time for the A* algorithm to compute the optimal path is 0.015 seconds, and the memory space is drastically improved, storing only the optimal path in memory in contrast with storing all discovered-extendable paths.

(1) a_star_optimized algorithm in Python:

```
def a_star_optimized(matrix, start, end):
  l = [start]
  output = []
  b = False

  while True:
    current_node = l.pop(len(l)-1)
    if not (current_node in output):
      output.append(current_node)
    else:
      continue
    if current_node == end:
      b = True
      break
    neighbours =
get_neighbours_w_weights(current_node)
    for i in range(len(neighbours)):
      neighbours[i] = [neighbours[i][0],
                       neighbours[i][1],
                       H[neighbours[i][0]]]
    for n in neighbours:
      if n[1] + n[2] == H[current_node]:
        l.append(n[0])
        break
    if not l:
      break

  if b:
    return output
  else:
    return None
```

**This algorithm uses the array H being the heuristic cost function populated as described above, and the correspondence matrix M, which stores for each two nodes i, j M[i][j] = weight of the edge connecting i to j, if i → j exists, otherwise M[i][j] = 0.*
**get_neighbours_w_weights code:*

```
def get_neighbours_w_weights(node):
    neighbours = []
    for j in range(len(M)):
        if(M[node][j] != 0):
            neighbours.append([j, M[node][j]])
    return neighbours
```

III. Utilizing The K-means Algorithm In Order To Avoid Traffic

The progress done so far is great, however distance is just one part of the equation here, as in finding an optimal path between two nodes on a map many other factors should be taken into consideration, and the most influential factor of all is without any doubt: traffic. The main difference between distance and traffic is that, distance between any two points on the map is rarely changed, and thus we can assume that it's static with a very low probability of change over time. In contrast, traffic is a dynamic factor that changes every several minutes, so if we were to include traffic in our heuristic function calculations to have it reflect not only distance, but traffic as well (which implicitly define the speed at which one may drive through this lane), so as a result we will be calculating time (as a function between distance and speed). This will result in a very poor performance, as we will need to recalculate the heuristic function every couple of minutes to reflect the current changes on the roads (concerning traffic information). That's the exact reasoning as of why we separated distance (static parameter) and traffic (dynamic parameter). The above defined function for the heuristic cost distribution for a given destination could be further generalized to reflect the distribution of the heuristic function for all possibles destinations/nodes.

Having done that hugely expensive step once, it will remain static and we use its results (unless updates occur, e.g., distance between nodes modified, new connections added, etc...). As for the traffic, the idea is that we need a much more performant algorithm (as it will be necessary to rerun or modify this algorithm multiple times) to quickly organize nodes into groups, each having a degree of traffic + distance level.

K-means reveals itself as one of the best candidates for such tasks, as it is an unsupervised machine learning clustering algorithm whose goal, given a constant k, is to divide points into k groups, called clusters. That is achieved by assigning to each cluster a center point, called centroid. After being initialized randomly on a given f-D space, f being the number of parameters (in our example we have distance and traffic as our parameters, so we will be working in a 2D space), at each iteration: (1) each node is assigned to the group of the closest centroid to its position, (2) centroids are repositioned as the center of all current nodes in their group, (3) repeat until convergence (i.e., centroids positions and groups remain unchanged). Fig.5,6,7 illustrates the iterations of a simple example of the K-means algorithm being applied on 39 points in a 2D space with k = 2.

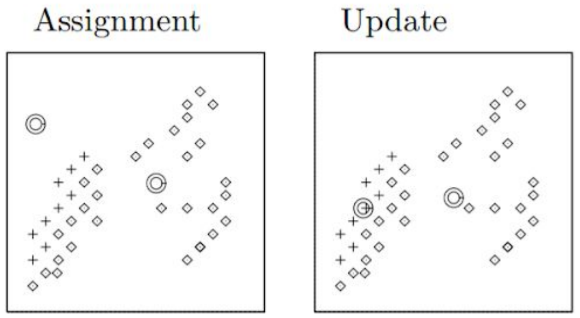

Fig.5 K-means Example Iteration 1 - "The two centroids are initialized randomly in the space,

and the shape of each point illustrates its belonging to each of the centroids"

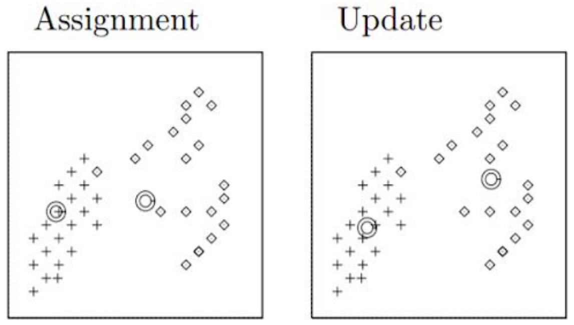

Fig.5 K-means Example Iteration 2 - "After repositioning the two centroids such as they represent the center of all points belonging to their group, the belonging of each point is reevaluated in accordance to the new positions of the centroids"

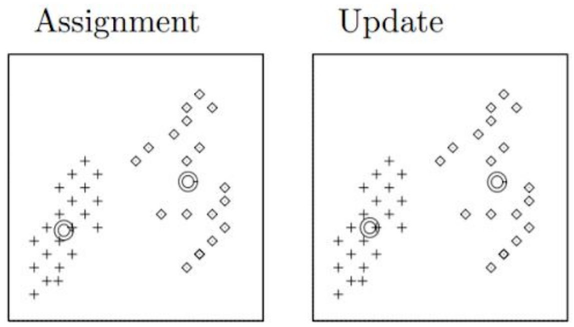

Fig.6 K-means Example Iteration 3 - "A slight update occurs in this iteration, as only one point got reassigned to another group, and thus this will be the last iteration of this example as in the next iteration the centroids positions won't be modified and the stop criterion will be met"

Fig.5,6,7 are snippets of the K-means algorithm course under the Cambridge University Press 2003 Copyright, in the section "An Example Inference Task: Clustering" which can be found in the following reference [CU001], along with a more detailed explanation of the algorithm.

After introducing the concept of traffic in our program, which is in fact a random number between 0 and 100 for each node stored in an array "cars", we proceed by normalizing both traffic and the heuristic function so that both are in the [0..1] value range (this will lead to an even influence on the clustering of the nodes), and we set k =3, with the spirit that the nodes shall be split into 3 groups: one low traffic/short distance group, one high traffic/far distance group, and a third in between group. Fig.7 shows the output of the K-means algorithm applied to our example for a random destination and traffic distribution.

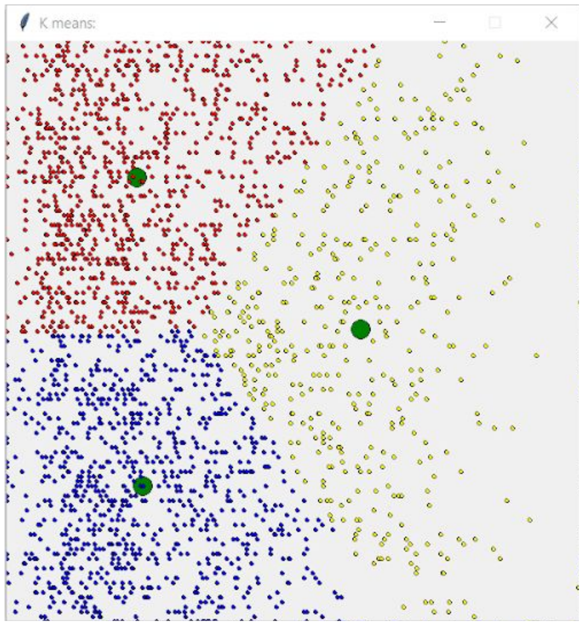

Fig.7 K-means Output Example - "Each of the green points represent a cluster's centroid. The blue points are in the "best" choice group having the best ratio of distance/traffic, followed by the yellow points and finally red points being the worst choice of nodes "

Having such a result in hand, we can proceed by modifying our a_star_optimized algorithm (that picks the next node n whose g(n) + h(n) = h(current_node). I.e., the lowest cost function, f(n), between the choice of next nodes) in the following matter: after collecting the possible next hop nodes, we don't directly pick the node with the lowest cost function, but rather we pick the node with the lowest cost function from the lowest K-means group. I.e., we seperate the possible next hop nodes into 3 groups depending on their K-means group belonging, then if there are any nodes in the first group (blue), we pick the node with the lowest cost function in that group, if not, we proceed to the second group (yellow), and only when none of the first two groups have any members, we proceed to pick the node with the lowest cost function from the last group (red). Note that to ensure convergence and to avoid infinite loops between two "blue" nodes, we must add the following condition to extend node n' to node n: h(n) < h(n') (i.e., ensure that we are always getting closer to the destination). Let's further explain this idea with the following example: Given a path to extend: π_3 = A b c d, the neighbours of d are collected in an array neighbours = [e, f, g, h], with the following heuristics: h(e) = 18, h(f) =27, h(g) = 35, h(h) = 31. We then proceed by splitting these nodes depending on their K-means group belonging, and will be stored in a single multi-dimensional array: groups; groups[0] = [], groups[1] = [f, h], groups[2] = [e, g]. The previous a_star_optimized algorithm would choose node "e" as its next hop node, however in the currently available information we can see that node "e" is in the third group, which means it must have a relatively high traffic level, which would result in it being a poor decision as drivers will get stuck in traffic and thus ending up losing a lot of "time" due to an algorithm that only optimizes "distance". However the a_star_optimized_with_kmeans algorithm will proceed as follows: (1) Check if any nodes belong to groups[0] := False. (2) Check if any nodes belong to groups[1] := True. (3) Pick from groups[2] the node with the lowest cost function → f. (4) Add node f to the path (π_4 = A b c d f), break and continue with node f.

(2) k_means algorithm in Python:

```
def k_means(k):
    centroids = []
    prev_centroids = []
    k_groups = []
    for i in range(k):
        centroids.append([uniform(0, 1),
                          uniform(0, 1)])
        prev_centroids.append([-1,-1])
    while not convergence(prev_centroids,
centroids):
        for i in range(k):
            prev_centroids[i][0] =
centroids[i][0]
            prev_centroids[i][1] =
centroids[i][1]
        k_groups.clear()
        for i in range(k):
            k_groups.append([])
        for i in range(len(nodesID)):
            min = euclidian_dist(
                [H_normalized[i], cars[i]],
[centroids[0][0], centroids[0][1]])
            min_centroid = 0
            for j in range(1, k):
                temp = euclidian_dist(
                    [H_normalized[i], cars[i]],
[centroids[j][0], centroids[j][1]])

                if temp < min:
                    min = temp
                    min_centroid = j
            k_groups[min_centroid].append(i)

        for i in range(k):
            if(len(k_groups[i]) == 0):
                continue
            x, y = mean(k_groups[i])
            centroids[i][0] = x
            centroids[i][1] = y
    return k_groups, centroids
```

(3) a_star_optimized_with_kmeans algorithm in Python:

```
def a_star_optimized_with_kmeans(matrix,
start, end, k_groups):
    l = [start]
    output = []
    b = False
    while True:
        current_node = l.pop(len(l)-1)
        if not (current_node in output):
            output.append(current_node)
        else:
            continue
        if current_node == end:
            b = True
            break
        neighbours =
get_neighbours_w_weights(current_node)
        for i in range(len(neighbours)):
            neighbours[i] = [neighbours[i][0],
                             Neighbours[i][1],
                             H[neighbours[i][0]]]

        groups = []
        for i in range(3):
            groups.append([])

        for i in range(len(neighbours)):
            for j in range(len(k_groups)):
                if neighbours[i][0] in
k_groups[j]:
                    groups[j].append(i)

        for i in range(len(k_groups)):
            if groups[i]:
                min = 999999
                min_node = -1
                for j in range(len(groups[i])):
                    temp =
neighbours[groups[i][j]][1] +
neighbours[groups[i][j]][2]
                    if temp < min and not
(neighbours[groups[i][j]][0] in output) and
neighbours[groups[i][j]][2] <
H[current_node] and
neighbours[groups[i][j]][2] != -1:
                        min = temp
                        min_node = groups[i][j]
                if min == 999999:
                    continue

l.append(neighbours[min_node][0])
                break
        if not l:
            break
    if b:
        return output
    else:
        return None
```

IV. Results

All test cases and CPU time performances shon in the previous sections have been conducted on an HP Envy 17" with a Core i7 CPU having the following specifications:

Intel(R) Core(TM) i7-5500U CPU @2.40GHz

Base speed:	*2.40 GHz*
Sockets:	*1*
Cores:	*2*
Logical processors:	*4*
Virtualisation:	*Disabled*
Hyper-V support:	*Yes*
L1 cache:	*128 KB*
L2 cache:	*512 KB*
L3 cache:	*4.0 MB*

Note: in all of the "K means" plotting windows, the vertical axis represent the (normalized) traffic level values (bottom-up ↑) and the horizontal axis represent the (normalized) heuristic function values (left to right →).

As we are using random initial values for the centroids in our K-means algorithm, each execution will result in a different output (a suboptimal solution, i.e., a local minimum). Finding the global minimum of a K-means clustering is NP-hard, and a number of solutions were proposed to address that, however very few got accepted because of their impractical implementations, so most applications uses K-means with multiple restarts [SS002].

In our core example map and with the relatively small number of test cases conducted, we found the best results with clusters of the same form as in Fig.8. Furthermore, intuitively it seems that the closer "d" is to ~0.15 (as the horizontal axis values ranges between 0 and 1), and the closer γ is to ~110° with $\alpha \doteq \beta$, the better the A* algorithm with K-means performs.

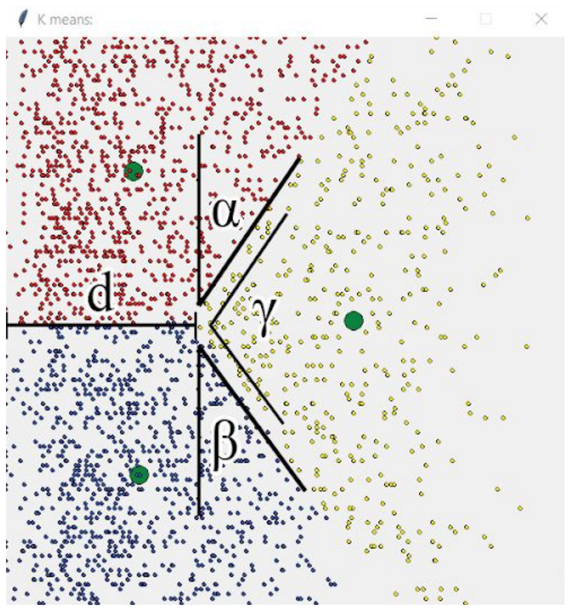

Fig.8 K-means Output Example - "d is defined to be the distance from 0 to the first node belonging to the second group. α and β being the angle between the vertical and the line separating nodes from the second group with nodes from the third and first group respectively. γ being the angle between the lines separating each of the first and third groups with the second group"

I.e., a low heuristic cost (that is between 0 and 0.15) won't affect the algorithm's choice, and only the traffic level will be influencing the nodes group belonging. After that threshold, nodes from the first and third groups starts to get separated by the second in-between group in a linear manner until it reaches another threshold, after which all nodes belong to the second group. Of course that still needs to be proven, which is part of the future work of this paper.

In the following example we will discuss in detail the output difference between the two algorithms, a_star_optimized (A*O) and a_star_optimized_with_kmeans (A*OK). Fig.9 illustrates the different paths taken by both algorithms, with black nodes being unvisited nodes, blue nodes being the nodes of the path outputted from A*OK (optimal_path) and yellow nodes being the nodes of the path outputted by A*O (suboptimal_path) and that are not taken by the path of A*OK (i.e., yellow nodes = suboptimal_path - optimat_path = x: x ∈ suboptimal_path and x ∉ optimat_path), and green nodes being the origin and destination nodes.

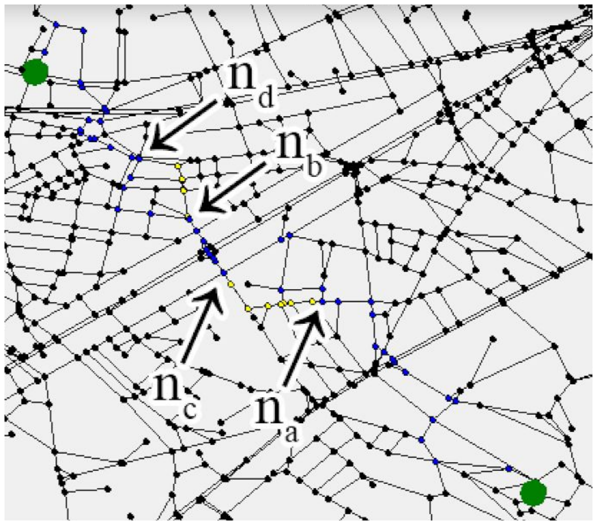

Fig.9 Example of Paths Taken by A*O and A*OK - "n_a and n_b are the nodes at which both algorithms' path got separated. n_c and n_d are the nodes at which both algorithms' path got reunited"

We can observe that when arriving at node n_a, two possibilities are available: either go left or go up. A*O chose the node that's at the left of n_a (which means that's the node with the lowest cost function), however A*OK chose the node on top of n_a, because this node is on a lower K-means group that the node on the left, despite having a higher cost function. Continuing with the paths we can see that they reunite at node nc, with A*O taking a path π_i' = A … n_a left … n_c, and A*OK a path π_i = A … n_a top … n_c. At node n_b we observe a similar situation, where algorithm A*OK opted to go left instead of up, in order to avoid traffic present at the node on top of n_b. As a result, A*O's path length is 2333.23 meters and A*OK's path length is 2705.4 meters. However, A*O's path would take 11 mins 8 secs to traverse, as for the A*OK it would only take 9 minutes 31 seconds to traverse its path. This in fact confirms our algorithms behaviour, as A*O is supposed to output the optimal "shortest" path, however A*OK's goal is to output the optimal "quickest" path.

Fig.10,11,12,13 illustrates more examples on the different paths taken by both algorithms. In those figures, blue nodes represent unvisited nodes, red nodes represent nodes in the A*OK's path, yellow nodes represent nodes present in A*O's path but not present in A*OK's path, and green nodes represent the origin and destination nodes.

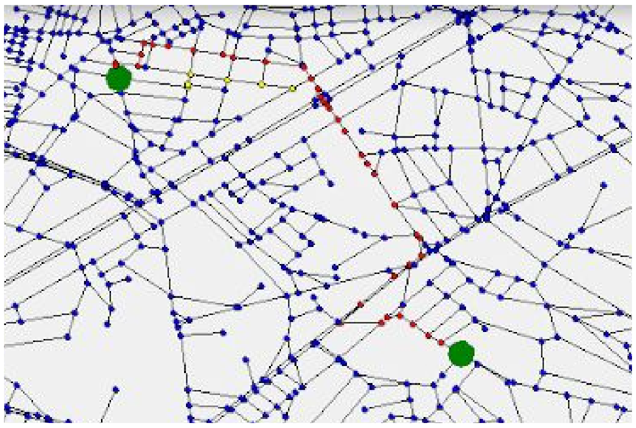
Fig.10 Example of Paths Taken by A*O and A*OK - "A*O's path is of length 1626 meters and would take 6 mins 44 secs to traverse. A*OK's path is of length 1652 meters and would take 5 mins 50 seconds to traverse."

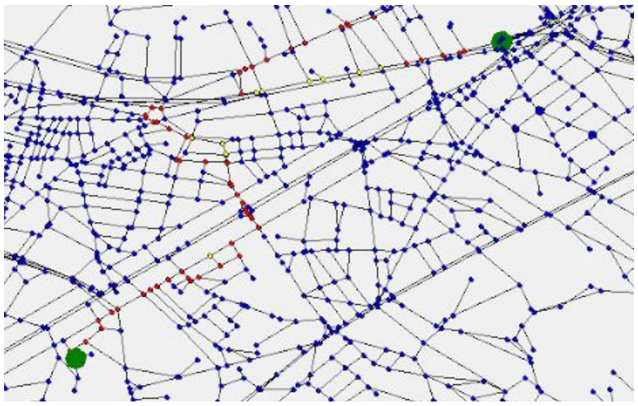
Fig.11 Example of Paths Taken by A*O and A*OK - "A*O's path is of length 2374 meters and would take 14 mins 30 secs to traverse. A*OK's path is of length 2765 meters and would take 10 mins 36 seconds to traverse."

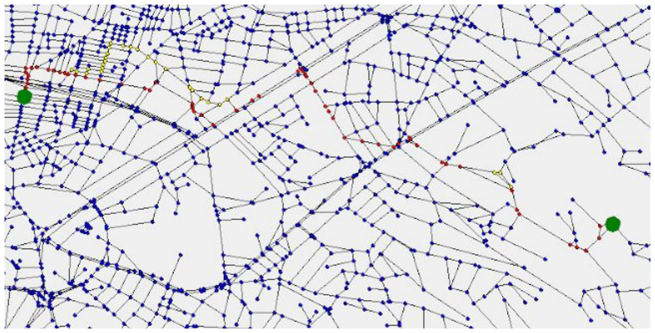
Fig.12 Example of Paths Taken by A*O and A*OK - "A*O's path is of length 3057 meters and would take 16 mins 30 secs to traverse. A*OK's path is of length 3101 meters and would take 15 mins 46 seconds to traverse."

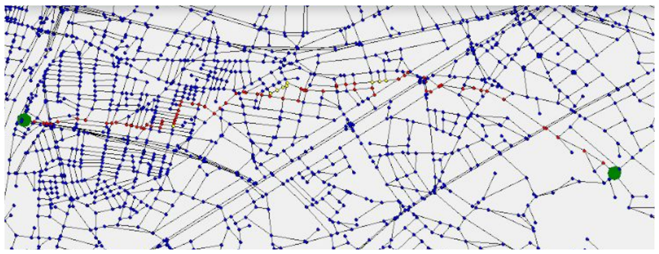
Fig.13 Example of Paths Taken by A*O and A*OK - "A*O's path is of length 2795 meters and it would take 14 mins 41 secs to traverse. A*OK's path is of length 2950 meters and would take 11 mins 52 seconds to traverse."

Nevertheless, having a completely random attribution of traffic levels for each node independently isn't what we'd call ideal or "realistic". One node having a low traffic level might be preferred over another having a higher traffic level, however this doesn't take into consideration the traffic of the following nodes. This behaviour resulted in moderate inconsistencies in our model. Several attempts have been made to try to achieve a more "realistic" distribution of traffic levels that isn't entirely based on randomness. One of those attempts was to calculate the traffic value of each node (after the first random initialization) as an average of degree 2 of all of its neighbours (nodes reachable from this node) and nodes from which this node can be reached. However this attempt failed as it resulted in a much more condensed range of values of traffic levels (as high values were lowered by the average and low values were raised by the average), which can be observed in Fig.14. Another attempt was to repeat the following process several thousand times: (0) initialize traffic values for all nodes at 0, (1) pick a random node n on the graph, (2) pick a random integer m in the range [5..20], (3) randomly extend the path originating from node n for m nodes, (4) increment the traffic value of all nodes in the resulting random path with a random number in the range [1..5]. This attempt tries to simulate real cars behaviour on the road, starting at a random position and taking a random path before eventually arriving at a destination. However this attempt also failed as it resulted in a much more rigid plotting and much more discrete values for the traffic levels, which is illustrated in Fig.15.

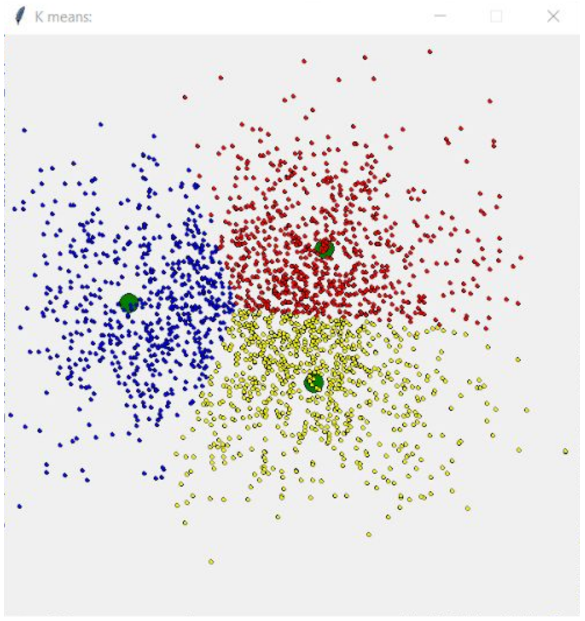

Fig.14 K-means Output Example for Attempt 1 - "We can observe the convergence of the nodes, as the traffic levels are now the average of degree 2 of their reachable and reachable-from neighbours. This results in a very bad clustering that is not stable with the nodes forming a circle shape, so clusters could be mode at any given angle depending on the randomly initialized centroids."

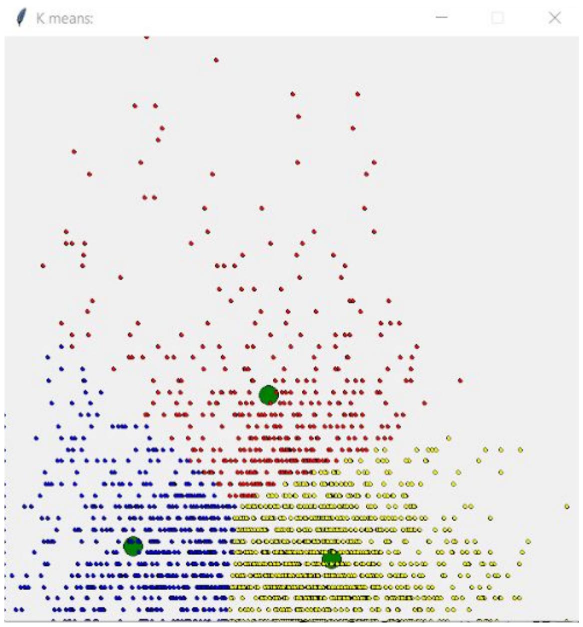

Fig.15 K-means Output Example for Attempt 2 - "In this plotting we can observe the "harsh" distribution of traffic levels, with few nodes having exceedingly high traffic levels when compared to the rest of the nodes."

V. Conclusion and Future Work

In conclusion, this work has helped in developing a sense of analytical thinking, presenting a new perspective on graphs and informed search algorithms. Throughout this work, processing performance and output performance has been optimized, and the difference in handling static (to some degree), and dynamic parameters has been greatly highlighted in order to further enhance processing optimization. Also, this work has greatly showcased the importance of parametrization and flexibility in programming which is, in my view, one of the most fundamental characteristics of any software. This characteristic will allow us to (relatively) easily further upgrade our algorithms and test many other test cases as described in the following paragraph.

An aperture of this work, is to study the difference in output performance for different "k" values for the K-means algorithm, which may lead to interesting results, subject to be analysed and intuitively explained (E.g., as explained in Fig.8). Such analysis or explanation may be quite hard for the bere eye to catch for such a general case scenario, however with more testings and observations, this can be greatly assisted. Plus, as noted before, the K-means algorithm doesn't always converge to the global minimum, so this intuitive explanation will allow us to "manipulate" or greatly influence the final clustering, based on the initialization of the centroids (I.e., it will save the cost of restarting the K-means algorithm, as we have a clear shape as an objective). Another step that should be taken is the acquisition of real world traffic data, or at least come up with a relatively realistic traffic distribution algorithm, so we can further assert our results and analysis. One final possible future project would be to expand this project so that it includes a predictive machine learning algorithm that shall assist the current decision by predicting traffic levels on the nodes of the graph, based on previously collected data organized by years, months, period of the year, day of the week, hours of the day, etc ….

VI. References

[OX001] Oxford University "Graph Theory 1736-1936"
https://global.oup.com/academic/product/graph-theory-1736-1936-9780198539162?cc=se&lang=en&

[WK001] Wikipedia "Shakey the robot"
https://en.wikipedia.org/wiki/Shakey_the_robot

[WK002] Wikipedia "Dijkstra's algorithm"
https://en.wikipedia.org/wiki/Dijkstra%27s_algorithm

[WK003] Wikipedia "Shakey the robot"
https://en.wikipedia.org/wiki/Shakey_the_robot

[IJ001] IJCSNS "A*-based Pathfinding in Modern Computer Games"
http://paper.ijcsns.org/07_book/201101/20110119.pdf

[BK001] Berkeley "A* Parsing: Fast Exact Viterbi Parse Selection"
http://nlp.cs.berkeley.edu/pubs/Klein-Manning_2003_ViterbiAstar_paper.pdf

[RG001] Research Gate "A comparative study of A-star algorithms for search and rescue in perfect maze"
https://www.researchgate.net/publication/238009053_A_comparative_study_of_A-star_algorithms_for_search_and_rescue_in_perfect_maze

[SS001] Semantic Scholar "Investigation of the * (Star) Search Algorithms: Characteristics, Methods and Approaches"
https://pdfs.semanticscholar.org/831f/f239ba77b2a8eaed473ffbfa22d61b7f5d19.pdf

[SS002] Semantic Scholar "Global K-Means (GKM) Clustering Algorithm: A Survey"
https://pdfs.semanticscholar.org/6154/a1ee06fbaf6961b4c4226fd5944fe761663d.pdf

[OM001] Open Street Map - Exact area studied in this paper
https://www.openstreetmap.org/#map=15/33.8893/35.5586

[SM001] SUMO - Documentation
https://sumo.dlr.de/userdoc/

[CU001] Cambridge University "An Example Inference Task: Clustering"
http://www.inference.org.uk/mackay/itprnn/ps/284.292.pdf