M-Star Path Algorithm:
White Paper

Ben Masefield
October 10, 2018

Abstract

A new method is presented for finding a path connecting two locations in a network of connections. The approach is different from methods such as Dijkstra (or the popular A* approach) and, depending on usage, shows several benefits over other methods.

Though the approach does not guarantee that the best possible path is found in all cases, it provides a significant speed benefit under the assumption that the grid within which the path is to be found does not change frequently. The method allows for a step by step determination of the path without requiring that the full path be determined prior. As an additional benefit, it also allows determination of whether two locations are connected without calculating the path at all.

For cases where the speed is more important than the absolute accuracy of the path, this method shows significant benefits in both speed and accuracy over even the simplest algorithms such as Greedy (Best-First search) or similar approaches without the drawback of the possibility of the path becoming trapped by an enclosure. Determination of whether two locations are connected can be achieved by scalar comparison of one value within the matrix.

The descriptions within this document are based on the approach and information presented in US Patent No. 10,063,462 using a more narrative presentation. Any items not explicitly presented in the patent application should be considered a logical derivative of that work.

Introduction

Humans - and most animals - have an instinctive approach to finding the shortest path from one point to another. This getting 'from A to B' finds its use in a very diverse set of circumstances such as reading a map to determine how to navigate through streets to arrive at a destination or drawing a line in a labyrinth from the entrance to the center in a common brain teaser.

A similar requirement frequently arises with robots and computers. However, there is no intuitive approach that can be formulated in a way that computers understand. A tempting consideration may be to use brute force and have the computer simply calculate all possible paths and then choose the shortest one. This is often how some problems are resolved but for something as complicated as finding a path, there may be simply too many possible combinations to find the best route in a reasonable amount of time.

The most straight forward approach is to generate the path using whichever path segment most directly points towards the destination. For some problems, this is a perfectly acceptable solution. The downside with this is that if there is some form of obstruction between the start and destination, then the approach will incur a large penalty in finding its way around the obstruction, not only in the calculation of the path but also in the final path chosen.

A better solution to this is Dijkstra's algorithm which finds the shortest path between a set of locations. The algorithm is relatively easy to implement and always finds the (or one of multiple) route

that is the shortest out of all possible routes. One of the greatest downsides of this algorithm however, is that it essentially requires the computation of any possible neighboring locations to each location along a possible path. This is not of great concern for cases where the route nicely goes from one location to another where the number of additional computations remains fairly low (basically the immediate neighbors to the final route), but can be quite problematic in cases where there is some form of barrier that needs to be avoided. In this case the route becomes trapped and only by continuing the computation sideways can the final path be found (essentially by back-tracking) and this effectively wastes computation time on items that were not of further use. Other algorithms have been proposed and are frequently used that take the essence of Dijkstra's algorithm but add methods by which only those paths are considered that are likely to lead to the destination.

Though the concept is quite trivial, one additional use for these algorithms is sometimes to simply find whether there is any path at all. In other words, the length of the path and the number of stops along it are of less concern, rather the question becomes whether there is any path through which the destination can be reached.

The algorithm presented in this paper takes the scenarios to which Dijkstra's algorithm (and algorithms derived from it) are applied and approaches the solutions from an entirely different perspective. Though existing algorithms are very useful in their current form, in some circumstances a better approach can be used. This novel algorithm shows some interesting properties in terms of efficiency and speed. The remainder of this document is going to discuss the approach of this algorithm and how it applies to the path and connectivity finding problems as well as to present a few different applications.

Part 1: Basics

The mathematical specifics relevant to obtaining the matrix used for the calculation of the steps is not discussed here. For the purposes of this discussion It should suffice to say that the M* algorithm solves, at its core, an equation of the form $M^* \times k = s$. The matrix M* is obtained from the locations and their attached connections. It is of size N×N (where N is the total number of locations -or nodes). Its size is independent of the number of connections between these locations.

In terms of generation, the M* matrix has an approximate time complexity of O(N*C) for the first step (where C represents the average number of connections at each location and could thus be considered a constant). The second major step in obtaining the final matrix is a (sparse) matrix inversion and thus has a time complexity of $O(N^3)$ (or slightly better for algorithms other than Gauss–Jordan elimination). This overall high complexity is the main reason why the M* algorithm should not necessarily be considered a general-purpose algorithm for path finding as other methods will generally outperform it by a fairly large margin. Speed improvement can be achieved by applying parallel processing to the matrix inversion - though that topic is outside of the scope of this document. It is worth noting however, that a majority of other path finding algorithms have issues with taking advantage of parallel processes as the path needs to evolve based on knowledge of the prior state. As such, if a sufficiently parallel implementation can be obtained, this approach may even surpass other algorithms in terms of final computation time, even for a frequently changing grid.

What is to be specifically noted, is that, once the matrix inversion has been completed, the M* matrix is completely re-usable for any combination of start and end locations, in other words, the M* matrix is dependent on the nodes and their connections but not the starting or destination locations – thus the matrix is not simply the accumulation of computing all paths between two points.

Part 2: Finding an example path

Here we will take a look at what is involved in finding a path from one point to another in an example grid (or a mesh) of nodes. The details about the underlying mathematics will be kept brief for the sake of simplicity in the explanation.

Each connection within this mesh has an equivalent to a distance of that section (it could easily also be some other parameter such as cost, speed, etc.) that we wish to minimize over the final path. The example mesh is shown in Figure 1:

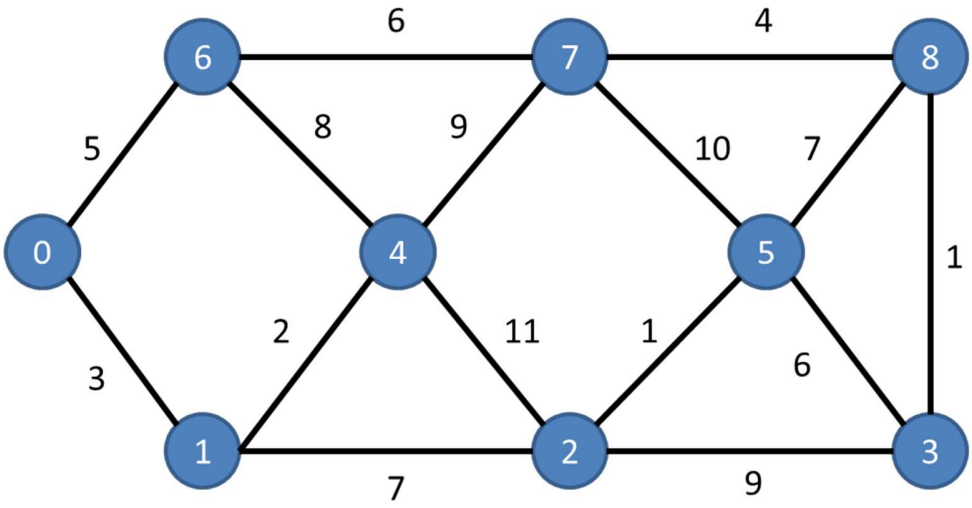

Figure 1: Example Mesh

Here the nodes are identified with a blue circle. Each node has an identifying number associated with it given in white. The order is not of importance. Each connection must be between exactly two nodes but each node may have any number of connections. The 'distance' is given as a value next to its associated line. The values shown do not reflect the graphical representation but rather a random value.

Going through the steps[1] of generating the matrix M* from the connections shown in Figure 1 results in the matrix:

$$M^* = \begin{bmatrix} 10.0 & 10.0 & 10.0 & 10.0 & 10.0 & 10.0 & 10.0 & 10.0 & 10.0 \\ 10.0 & 7.651 & 8.035 & 8.15 & 7.974 & 8.083 & 8.916 & 8.321 & 8.174 \\ 10.0 & 8.035 & 5.077 & 5.74 & 7.571 & 5.354 & 8.274 & 6.731 & 5.879 \\ 10.0 & 8.15 & 5.74 & 3.932 & 7.605 & 5.428 & 8.083 & 6.142 & 4.482 \\ 10.0 & 7.974 & 7.571 & 7.605 & 6.739 & 7.585 & 8.376 & 7.656 & 7.612 \\ 10.0 & 8.083 & 5.354 & 5.428 & 7.585 & 4.753 & 8.194 & 6.485 & 5.548 \\ 10.0 & 8.916 & 8.274 & 8.083 & 8.376 & 8.194 & 6.807 & 7.798 & 8.043 \\ 10.0 & 8.321 & 6.731 & 6.142 & 7.656 & 6.485 & 7.798 & 5.262 & 6.019 \\ 10.0 & 8.174 & 5.879 & 4.482 & 7.612 & 5.548 & 8.043 & 6.019 & 4.149 \end{bmatrix}$$

[1] The generation of the matrix is described in the patent and contains proprietary steps that are not further discussed in this White Paper.

It should also be noted that, from a practical perspective, there really is no symbolic way to represent the inverted matrix. Inversion of matrices above about 5x5 can only really be done using actual numbers and algorithms that handle such an inversion.

Having generated the M* matrix, all the hard work has been done to find the shortest path from any possible start to any possible end location. Given a start location/node and a destination, the process consists of the following steps:

- First, find the two columns in the matrix that correspond to the start and the destination. The start column is multiplied by -1 and the destination by +1 and the two columns are added together. This only needs to be done once and can also be deferred until the individual values are actually being used, which is generally more efficient (and the approach used in this example).
- At the given location, find the connection with the highest calculated value which is found by dividing the difference of the row corresponding to the given location with the row corresponding to each neighboring location by the distance between the two of them.
- Having calculated all values at a given node, the next location on the path becomes that which is located on the connection with the highest value.
- This process continues until the next location is the destination.

As an example, and without spelling out the equations in too much detail, we will look at a case where the starting point is at node 0 and the destination is at node 5. The M* Path Finding approach looks at each neighboring[2] node and chooses the one with the highest value as the next location (for reference, the calculation of the path value is abbreviated as "start-location → end-location", the segment that results in the highest value is underlined):

At node 0, the two values are:
- 0→1: (10-10-8.083+10)/3=**0.639**
- 0→6: (10-10-8.194+10)/5=0.361

Thus node 1 is the next location at which there are three values (neighbors) to calculate:
- 1→0: (8.083-10-10+10)/3= -0.639 (note this is just the negative of 0→1)
- 1→2: (8.083-10-5.354+10)/7=**0.390**
- 1→4: (8.083-10-7.585+10)/2=0.249

So, node 2 is the next node. At this location:
- 2→1 = -0.390
- 2→3 = -0.008
- 2→4 = -0.203
- 2→5 = **0.055**

Which leads to the destination. Though this last step may seem trivial since the destination is one of the neighboring items, in some cases it may occur that it is shorter to travel via additional nodes than to directly go to the final node and thus the process needs to be completed until the path actually uses the destination as the current node.

[2] If the algorithm would simply look for the smallest distance at each location (equivalent to a Nearest Neighbor algorithm), then the path would be 0→1→4→6→7→8→3→5 with a total length equivalent to 30 units. This is clearly not the shortest path but would have the same time complexity in calculating the path.

Part 3: Finding a Connection

As mentioned, path finding is often times reduced to simply determining whether two nodes are connected at all. This can apply to a number of cases, such as determining whether one computer is connected with another by means of a network, whether there is a road from one location to another or whether two people share the same social circles. With a different approach this would be resolved by finding the shortest path, and if there is a result prior to exhausting all possibilities, then the two are connected. If the connection is very distant (has a high degree) then the computation can take a large number of calculations to arrive at a result.

For the M* Approach, the scenario is quite different. In fact, once the M* matrix has been computed, no calculations are necessary to determine whether two locations are connected! This is best shown through an example. Given a few separated networks as shown in Figure 2:

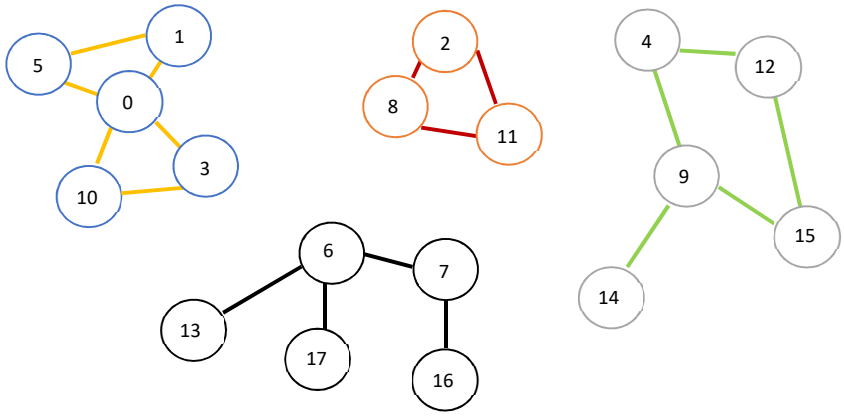

Figure 2: Separate Networks.

Here the distance between each set of nodes is simply considered as "1" but could be any value if there is additional information that is to be included. Going through the steps to generate the matrix, M* is found:

$$M^* = \begin{bmatrix}
199.8 & 200.0 & 0 & 200.0 & 0 & 200.0 & 0 & 0 & 0 & 0 & 200.0 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
200.0 & 199.5 & 0 & 200.2 & 0 & 199.9 & 0 & 0 & 0 & 0 & 200.2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 333.1 & 0 & 0 & 0 & 0 & 0 & 333.4 & 0 & 0 & 333.4 & 0 & 0 & 0 & 0 & 0 & 0 \\
200.0 & 200.2 & 0 & 199.5 & 0 & 200.2 & 0 & 0 & 0 & 0 & 199.9 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 199.6 & 0 & 0 & 0 & 0 & 200.0 & 0 & 0 & 199.9 & 0 & 200.2 & 200.1 & 0 & 0 \\
200.0 & 199.9 & 0 & 200.2 & 0 & 199.5 & 0 & 0 & 0 & 0 & 200.2 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 199.7 & 200.1 & 0 & 0 & 0 & 0 & 0 & 199.9 & 0 & 0 & 200.3 & 199.9 \\
0 & 0 & 0 & 0 & 0 & 0 & 200.1 & 199.5 & 0 & 0 & 0 & 0 & 0 & 200.3 & 0 & 0 & 199.7 & 200.3 \\
0 & 0 & 333.4 & 0 & 0 & 0 & 0 & 0 & 333.1 & 0 & 0 & 333.4 & 0 & 0 & 0 & 0 & 0 & \\
0 & 0 & 0 & 0 & 200.0 & 0 & 0 & 0 & 0 & 199.7 & 0 & 0 & 200.1 & 0 & 199.9 & 200.0 & 0 & 0 \\
200.0 & 200.2 & 0 & 199.9 & 0 & 200.2 & 0 & 0 & 0 & 0 & 199.5 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 333.4 & 0 & 0 & 0 & 0 & 0 & 333.4 & 0 & 0 & 333.1 & 0 & 0 & 0 & 0 & 0 & 0 \\
0 & 0 & 0 & 0 & 199.9 & 0 & 0 & 0 & 0 & 200.1 & 0 & 0 & 199.5 & 0 & 200.3 & 199.9 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 199.9 & 200.3 & 0 & 0 & 0 & 0 & 0 & 199.1 & 0 & 0 & 200.5 & 200.1 \\
0 & 0 & 0 & 0 & 200.2 & 0 & 0 & 0 & 0 & 199.9 & 0 & 0 & 200.3 & 0 & 199.1 & 200.2 & 0 & 0 \\
0 & 0 & 0 & 0 & 200.1 & 0 & 0 & 0 & 0 & 200.0 & 0 & 0 & 199.9 & 0 & 200.2 & 199.6 & 0 & 0 \\
0 & 0 & 0 & 0 & 0 & 0 & 200.3 & 199.7 & 0 & 0 & 0 & 0 & 0 & 200.5 & 0 & 0 & 198.9 & 200.5 \\
0 & 0 & 0 & 0 & 0 & 0 & 199.9 & 200.3 & 0 & 0 & 0 & 0 & 0 & 200.1 & 0 & 0 & 200.5 & 199.1
\end{bmatrix}$$

This form is perfectly valid, but it can be further simplified by considering any entry that is non-zero as one (1):

$$M^* = \begin{bmatrix} 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 1 & 1 & 0 & 1 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 1 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 0 & 1 & 1 & 0 & 0 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \\ 0 & 0 & 0 & 0 & 0 & 0 & 1 & 1 & 0 & 0 & 0 & 0 & 0 & 1 & 0 & 0 & 1 & 1 \end{bmatrix}$$

This is more efficient from a storage perspective since the data can be stored using bits rather than (multiple) bytes per entry.

In this form (or the previous), the matrix can be directly used to answer any questions regarding connectivity. This is best illustrated with an example:

- ***Is node 0 connected with node 5?***
  - Looking at the first row (for the first node since the nodes are labeled 0, 1, ...) and 6th column (for node 5 which is the 6th node), the value is 1 (or 200.0 in the earlier form), therefore there is a connection between the two.
- ***Is node 4 connected with node 5?***
  - Looking at row 5 and column 6, the value is 0, therefore there is no connection between the two.
- ***Is node 13 connected with node 16?***
  - Looking at row 14 and column 17. The value is 1 and therefore the two are connected.
- ***Is node 17 connected to node 0?***
  - The opposite is also true of course: we can look at the first row and last column just as much as we can look at the last row and first column to see that the two are not connected.
- ***How about finding all items that are connected to node 0?***
  - This is the same as simply looking at which columns are non-zero for the first row. It can be seen that this corresponds to the 1st, 2nd, 4th 6th and 11th corresponding to itself and nodes 1,3,5,10 as can be verified per the diagram.

It should be a trivial conclusion that the data shown in these forms of the M* matrix are symmetrical about its diagonal (axis) and thus the information contained in it can be reduced by almost 50%.

Since non-zero entries were reduced to values of one, the data is obviously no longer usable for path finding in this form.

Part 4: Avoiding Locations

Avoiding a location while travelling from one point to another seems like a fairly straight forward item. With other algorithms, this is accomplished simply by removing any locations to be avoided from the available mesh. With the M* approach, this is done slightly differently since its main benefit is re-usability of the computationally heavy M* matrix whilst keeping the path optimized[3].

The result is no different from what has been previously considered. The only difference is that additional terms are introduced in the calculation of the next location to account for all locations to avoid. The terms can be thought of a constant value added to the numerator. It is not accurate to consider that the path will always avoid those locations since it is still possible for a path to pass through the same nodes. A more certain form of the algorithm would need to additionally account for this case and selectively skip these locations during the calculation of the next-best location, most likely by having a hashed look-up table where these locations are identified.

Though this seems somewhat complicated, there is nevertheless the benefit of a rapid re-use of the matrix for scenarios that change more frequently.

Part 5: Applications

The M* approach has a number of uses in its current form, some of which are briefly discussed in this section. It is the intent to provide an overview of scenarios where the method can be useful without being too thorough on how the implementation would be applied. This is intended as a starting point for a more in-depth discussion on how these may be achieved.

Standard Path Finding

The most obvious use is to find a path from one location to another on a map. This would be applicable to things such as GPS navigation and similar. The same idea is also often used in computer games where an AI opponent is moving towards a target while avoiding structures and similar. For this application, the method is particularly useful since the mesh (the floor plan or similar) rarely if ever changes and both the opponent and/or target can move frequently or there are a large number of characters moving at the same time. The approach generally yields a path to the closest destination if there are multiple destinations which would be useful in a scenario where opponents are part of a group and they were to individually seek out the closest opponent.

The approach also allows only a single step to be calculated (i.e. should the character turn right or left at an intersection) without requiring the entire path to be plotted and as such requires far less computation than an algorithm that needs to compute the entire path each time. This is useful for cases where the start/destination may frequently change as the entire path will likely not be followed but rather only one step is taken before re-evaluating whether the current path is still the optimal.

[3] A simpler form would simply skip any locations during the determination of the next location. Though this is possible, it is not the most optimal approach but could be sufficiently accurate in some scenarios. The danger with this approach is that the path becomes trapped by avoidance locations.

One further scenario where this step-by-step option may be useful is for an incremental approach to using GPS direction as the direction to take can be made available immediately (should the user deviate from the path) while at the same time a different algorithm may be calculating the entirety of the path under consideration of other factors.

Connection Determination

The second use, connectivity finding, may also be used in generating a path since it allows a determination up front about whether two locations can be connected at all. Thus, when a GPS route is planned, a first step may simply be to determine whether it is possible to provide such a path rather than to wait for the calculations to reach a point where they determine that no such path is available.

A different use might be found in social media type of applications where users are connected with each other or data is only to be shared with people of the same social circle or within a social circle that is limited by the levels of associations.

Obstacle Avoidance

The location or obstacle avoidance is similarly useful in path finding but likely more tailored towards autonomous guidance systems where limited information is available at any given time. For a robot to move through a location while avoiding personnel or other units, frequent updates to what obstructions are present is necessary while still having a set destination to travel to. For such cases, the system matrix may simply represent a rectangular mesh with nodes at a regular interval and connections from each node to its neighboring nodes. This would represent a Cartesian grid in two or three dimensions where the spacing is simply the resolution of the sensor output. Having a uniform matrix like this would allow it to be hard coded into the robot's control software and the only items that require updating would be the avoidance locations and the destination (the current location within the mesh would be thus considered the starting location). This approach could even be moved into hardware directly with the matrix and its functions designed into a dedicated hardware component (or programmed in a FPGA, ASIC or similar).

It should also be mentioned that the algorithm is in no way limited to being in two-dimensional space. The exact same approach can be used for a 3D mesh (or higher dimensions if that makes sense) where different levels can be accessed. This may be applicable to robots navigating through a maze while changing levels or moving a robotic arm in such a way as to avoid obstacles (and perhaps applying path finding incrementally for each joint to obtain the most efficient motion). The matrix and its generation follow the exact same rules for 3D cases and the matrix still only has a size N×N (for N locations), irrespective of the number of connections between them or the dimension of the coordinates associated with each location. The difference is that the greater the number of internal connections, the less sparse the matrix becomes.

For the case of an autonomous vehicle moving within a changing environment, a scenario may be encountered whereby the next closest charging station is being sought and a number of obstructions are placed in its way. By utilizing this method, the direction to move in is always towards the next closest available station.

A different use for the obstacle avoidance may also be found in an implementation for a packet routing protocol where nodes may frequently be taken offline. In its normal form, the M* approach is well suited to being a packet routing algorithm running on the backend (replacing Dijkstra in an IS-IS or OSPF protocol for example) but would bring with it the ability to easily modify the underlying matrix (which may be updated by a master router in the network and made available to individual routers) based on which

nodes are available and which are not (information about unavailable nodes may be provided by their neighboring routers/nodes if they do not respond to a request within a timeframe). Specific implementations would obviously need to further define the function of the protocol and whether it makes more sense to use a centralized database or a different approach.

In electronics design software (ECAD), it is a frequent requirement to auto-route connections between components in a circuit board. Using an approach such as this, an iterative approach can be used whereby routes are determined using knowledge about what locations have already been occupied by other components or traces. Though the approach would still need to be iterative and consider different pins as starting points if it becomes impossible to complete the connections based on boundary conditions. As the approach does not require modifications for 3D path finding, having any number of layers through which a conductor trace can be laid becomes a trivial extension. The spacing within the grid would then be determined by the resolution to be used and likely a factor of the minimum trace width and spacing.

In a Computer Aided Manufacturing (CAM) scenario, an approach like this may be used to provide a 'skimming' route to allow a cutting tool to move from its current location to the starting location of the next feature being machined without requiring the lengthier move to a safe height above the part only to then be lowered again once the new location has been reached. This would effectively reduce the time that the cutting tool travels without performing actual work while still safely avoiding collisions with existing material or machinery. Definition of what locations are safe to move through and which are not would come from performing an interior/exterior comparison between an arbitrarily coarse volumetric grid and the component being machined based on where it currently is in the program. Currently, cutting tools frequently have a very rigid motion when moving from one location to the next, between performing actual cutting. This is mainly due to the concern that the cutting tool or its holder may collide with material that is not to be cut or other components of a similar nature. Using a better path finding approach between these moves would reduce the overall time that the program takes to complete a manufacturing operation.

Multiple Destination Route Planning (Advanced TSP)

The idea of how to traverse multiple points of interest in the most efficient manner is often referred to as the "Travelling Salesman Problem" (TSP). Within the TSP, the problem to be solved can be considered to be of two levels of complexity. The first, the geometric problem, is to look at the locations to be traversed and to find the best order in which they are to be visited in order to minimize the overall round-trip length. The distance from one location to the other is thus simply the Euclidean distance of the vector between them.

In order to solve this, a few different approaches are commonly used with varying results in terms of their computational time requirements and accuracy of getting the shortest trip. The first and in many ways easiest is the Nearest Neighbor or Best-First Algorithm, also known as the Greedy Algorithm, which simply looks for the location that is the closest to the current location in order to determine the next item. This approach is still very common in use with Computer Aided Manufacturing (CAM) where hole drilling or feature machining is done by simply moving from one location to its nearest neighbor. In general, if only a few features are required, this approach is sufficiently accurate though it does lead to inefficiencies with larger numbers of points or when the nearest points start to lead the path away from other points that still need to be visited (a problem encountered in electronics manufacturing where PCB through holes

and vias can easily number in the hundreds to thousands and any time savings in the motion required to go from one to the next is multiplied by the number of boards being produced).

Another approach is a form of a minimum insertion approach which (usually) starts off by finding the convex hull or similar perimeter around the points and then iteratively adds remaining locations between two based on minimizing the effect of such an insertion.

Beyond these two (and a number of other approaches), the most efficient and generally best approach is to use what is known as a k-Opt algorithm, specifically that proposed by Lin & Kernighan which uses one of the less efficient methods to first generate a starting order and to then swap items in order to improve upon the overall round trip. The number of items swapped can change (thus the "k" in k-Opt) and it is generally considered the best heuristic algorithm for solving the TSP.

It should perhaps be briefly mentioned that for small numbers of locations to be visited (around 10-12, depending on the available computational power and time) a brute force solution can be used that finds the best combination of locations by attempting all possible combinations and to then use the one that is the shortest. This is however, not feasible for larger sets as the number of calculations grow as a function of N-factorial (N!) (so, for 5 locations there are 120 possible orders, for 8 there are 40,320 and for 10 locations 3,628,800 different combinations would need to be compared). Calculations with more than 12 locations also require consideration due to the permutations exceeding the (32-bit) integer count. For illustrative purposes, a total of 15 locations evaluated in this way, under the assumption that each evaluation takes 1 micro-second (an optimistic assumption), would still take a total of 15 days to complete. Common routing problems can easily number in the hundreds of locations.

The geometric problem however, where the order is determined by looking at the simple distance[4] between two points, is a simplification that does not hold in a real topological scenario (it is valid for pick and place, drilling operations and similar). If a delivery service is using a k-Opt algorithm to determine the order in which to deliver packages then any two locations that are close to each other from a geometric perspective - but may in fact be far apart if there is some dividing feature between them, such as a river or some other non-road space- would be incorrectly given an order that makes the overall route longer.

The only way that such an algorithm can be used effectively for the determination of the order of visiting a large number of locations is by also considering the actual travel distance of the path between the two locations, perhaps even taking into account elevation changes.

Though it is possible (and is likely useful even with the M* approach) to generate a cache (lookup-database) of distances for all possible combinations, doing so by using any of the common methods would be inefficient from a time perspective. Instead, the performance of the M* approach allows the distance between two locations to be computed on an as-needed basis and then stored in a cache for quicker retrieval as needed. Further, as the M* matrix is completely re-usable, it is possible to perform all of the optimization steps on a number of parallel processors with the matrix and cache data being shared. It may even be advantageous to first perform the route-distance calculations for all combinations (a total of N^2 determinations) on a highly parallel compute-platform and then to use those values in the k-Opt algorithm without calculating the geometric distance at all. These route-distances could be retained for later re-use as their storage requires very little space.

This approach, whereby a route for a large number of waypoints is obtained by first computing a simplified route (likely using a minimum insertion approach) and then -optionally- refined using a geometric k-Opt algorithm and finally applied on a street level by using actual paths rather than geometric

[4] The calculation is usually to use the square of the distance rather than the actual distance as it improves the calculation time while still fulfilling all larger/smaller than evaluations necessary.

distances for the k-Opt is something that does not appear to be in current use, mostly due to the computational overhead demanded from other path finding methods during the stage where the actual paths are determined.

The most obvious use of this combined approach is with large delivery services that heavily rely on getting an efficient route planned, for a frequently changing set of locations, in order to minimize the total distance travelled. Similarly, though, almost all utility services provided by cities rely on having an efficient way to collect solid waste, distribute mail and similar items. Usually, this route is determined by a block system where the driver may be assigned a specific route and covers locations similarly to a Nearest Neighbor approach. Though this is reasonably efficient, problems may arise for cases where there are more or fewer delivery vehicles available, routes need to be combined or re-assigned or similar. For those cases a specific method by which the route can be minimized could quickly lead to very significant savings in operating costs. With all of these, the major hurdle to an improvement is simply the amount of data that needs to be processed which is where the M* approach can significantly improve upon existing methods.

General Notes

Though the algorithm shows some noticeable performance improvements, there are things to keep in mind. Some of these are briefly summarized here:

- The algorithm, though general in its form and use, is not always the best approach to solving a path-finding problem. Dijkstra (and A*) algorithms can likely find a path more quickly if the possible paths (the mesh or grid) is changing frequently or the problem only solved a small number of times. They also (generally) guarantee that the path is the shortest/best possible, with the M* approach, this is not necessarily guaranteed as it depends on the grid being used in some measure. Speed improvements for changing grids with the use of parallel processes can be obtained, as discussed.
- The numerical efficiency can be modified to better suit the circumstances. As shown earlier for the connectivity finding aspect, the data contained within each row/column of the matrix can be reduced to save memory/space. It is quite possible to perform a path finding approach using less than single precision (aka. 'Float'), even using Half precision (2 bytes per floating point value) or even single Byte level precision. Similarly, all values can be represented by integers making computation more efficient, especially on processors that are not fitted with a floating point (co-) processor. However, it is beneficial for the M* matrix to be generated using a higher level of precision (if possible) but can then easily be re-sampled or fit into other data types to allow for a more efficient handling of the information (though possibly at a loss of accuracy in finding the most efficient path).
- For cases where all the connections from A to B are equivalent to connections from B to A (so cases that don't have an equivalent to a one-way street) the M* matrix can be reduced in size to only consider the data on the diagonal and to one side and thus be represented in a linear array (vector) rather than a multidimensional array. The reduction in this case would be from N^2 to ($N^2/2 + N/2$) which approaches 50% for large matrices.
- Specialized cases with multiple starting locations and/or multiple destinations can be incorporated with ease without modification to the original setup or generation of M*. Equally, multiple 'path finders' can simultaneously be participating - though crowding of paths may occur unless they are assigned a priority and considered in turn, after which their path then becomes a set of weak

avoidance locations for subsequent path finders. This would still allow multiple users to pass through the same location but those locations would become increasingly less efficient (due to an increasing level of avoidance) as more users passed through them.

- The choice of the parameter is free to be chosen as any item that is to be either minimized or even maximized, provided that makes sense. Cost, fuel consumption, time, distance, speed and others are all possible parameters to be used but other items could easily take their place depending on the scenario.
- Once the computational complexity of determining the M* Matrix has been completed, determining the full path between two nodes has a time complexity of O(n*c) where n is the number of nodes along the final path and c is the average number of connections for each of those nodes. This is noticeably faster than Dijkstra which will in general run in $O(N^2)$ with additional time required to handle obstructions. A Greedy/Best-First search algorithm will generally run in O(n*c) too, but even a simple obstruction in its path will provide a non-optimal solution.
- In general, the path through the grid does not apply any special treatment to the selection of the next location at a given node, it merely follows the highest value. Under special consideration, it may be possible to quite easily modify this choice to add an additional term for each potential next node to account for real-world effects (specifically to delivery services and similar) whereby a 'left-turn' would receive an additional penalty compared to a 'right-turn' or similar. This penalty could be easily implemented on a per-location basis and would add a term to the calculations that would be positive (for straight-through or right-turn) and negative (for left-turn or similar). The magnitude would likely need some additional consideration in regards to what it represents in a real-world scenario.
- The matrix inversion process can be performed on parallel processors. As the process is relatively straight forward, it is possible to do this on a GPU cluster with some of the more sequential tasks being handled by a CPU. A paper titled "Matrix Inversion using Parallel Gaussian Elimination" from the University at Buffalo shows that even very large matrices (order >10,000) can be inverted in seconds. This approach would allow use of this algorithm to be used in a similar fashion in cases where the underlying grid needs to be modified more frequently.