The UBL Path-Matching Method

# Introduction

This document presents a detailed outline of the UBL Method. This document is intended to communicate the mathematical, computer science, and software engineering concepts required to produce the software model in a form that can learn and be queried. The document is intended as a specification for the development and extension of the UBL Method.

# Outline

1. Problem Description
   1. Learning Routes
   2. Classifying Transport Methods
   3. Storing and Finding Routes
2. High-level Solution
   1. Spatial Tree Structure
   2. Path Matching Method
3. An Example Case
4. A Mathematical Primer
5. The Green Points Algorithm
6. Input and Output
7. Mathematical Definitions
8. Software Architecture
9. Development and Testing
10. Risks and Assumptions
11. Conclusion

# Problem Description

To produce a model that learns repeated travel routes of participants. The routes should be classified according to person, movement type (walking, cycling, public transport, car), and time-of-day based on the observed feed of latitude and longitude data emanating from that person. The system should record the route based on a set of heuristics evaluated during participant observation. We may think of all routes as being stored into a **World Graph**. During learning we assign a set of weights or colours to the directed acyclic graph (**DAG**) produced by observing the motion in the World Graph.

The learning system must be able to classify the transport method. This is a non-trivial problem and will require the additional capture of the observed time at particular nodes in the route. This means an individual path must be captured, per person, per node, per trip (they may be averaged later). There should be support for tagging a route with pertinent information.

The model is learning routes continually as part of its data feed. The next part of the problem is to query the model for a route. The goal here is to specify a start location, an end location and to query the model to produce matching paths in the graph that are similar in terms of any specified criteria (e.g. similar route, both in a car, at the same time of day).

In order to perform spatial queries on the path data, a suitable method for storing and retrieving the data must be developed. Firstly, a suitable method of storing both spatial and path data must be developed. Furthermore, the algorithm should store this data in the most space-efficient manner and must allow rapid querying and traversal in order to match pathways. The graph must be built for rapid, parallel route evaluation using pathfinding algorithms.

Finally, the method should be able to evaluate a set of found paths for a best match based on criteria such as energy efficiency, time of day, and possibly other factors to be added later. In addition, the method should be able to calculate a cost contribution based on metrics derived from the modified path based on the original path, called the **Green Points Algorithm**. This will allow a fee-metric to be constructed on based factors such as additional time taken or fuel cost.

# A Description of the Solution

The solution proposed by this document can be explained by thinking of the world map being represented by an image. Suppose a user of the application is assigned a particular colour. Then we could represent the current location of the user by drawing a faint dot in the cell at which the user is located. Thus, if the user travels on the map, we trace the outline of where they travelled by drawing a dot wherever they have been located. We assume these dots to be connected and impose several heuristics for breaking the paths up into routes.

Suppose the user is recorded in the same cell again and again. Each time, we add a little more colour to the dot until it becomes very bright. We can then discriminate the highly active paths from the less active paths by determining the “brightness” of a path. Furthermore, each dot stores the time at which it was recorded allowing further derivation of useful statistics about the route:

* Average speed
* Travelled Distance
* Transport Method

Furthermore, the system can learn these data independently of Google Maps and with high degree of accuracy because it is based on real, sampled data rather than predefined networks. The UBL method also avoids the difficulty of matching vector paths in maps by simply not considering them at all. If a user travels every day from home to office in a car during rush hour, the system can determine the transport method by examining the location data and timing of cell transitions in the path. A person walking will produce very different path data to someone driving in a car or on public transport.

Suppose I wish to share a ride with someone going in approximately the same direction I’m heading. It would be intractable to compare all paths stored in the system to the current path being considered, so a method of spatial indexing is required. Spatial indexing allows us to query physical space for data stored there. It is a method of producing indices that rapidly resolve questions such as:

* How many paths start in a radius of **x**?
* Do the paths follow the same general route?
* What is the Manhattan distance between two cells?
* What is the displacement (or routing cost) between two paths?
* What is the cheapest routing cost between two points?

# High-Level Solution

The method can be roughly broken into three major components:

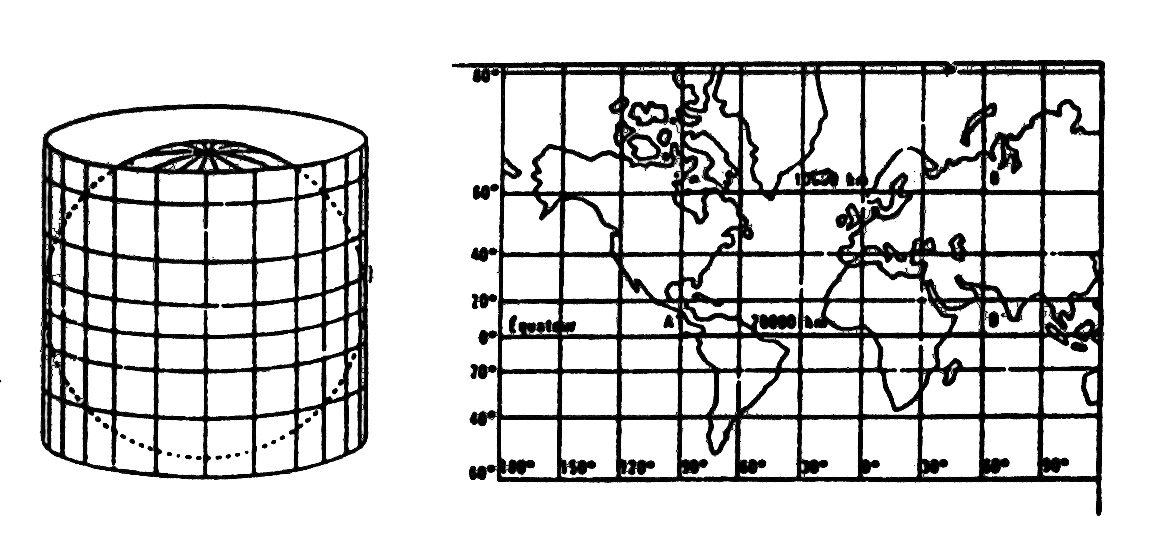
1. Learning Paths
2. Heuristic Optimisation
3. Path matching and quantification (green points).

The goal of the method is to compare the most-travelled routes of a person for matches with other people.

## Learning

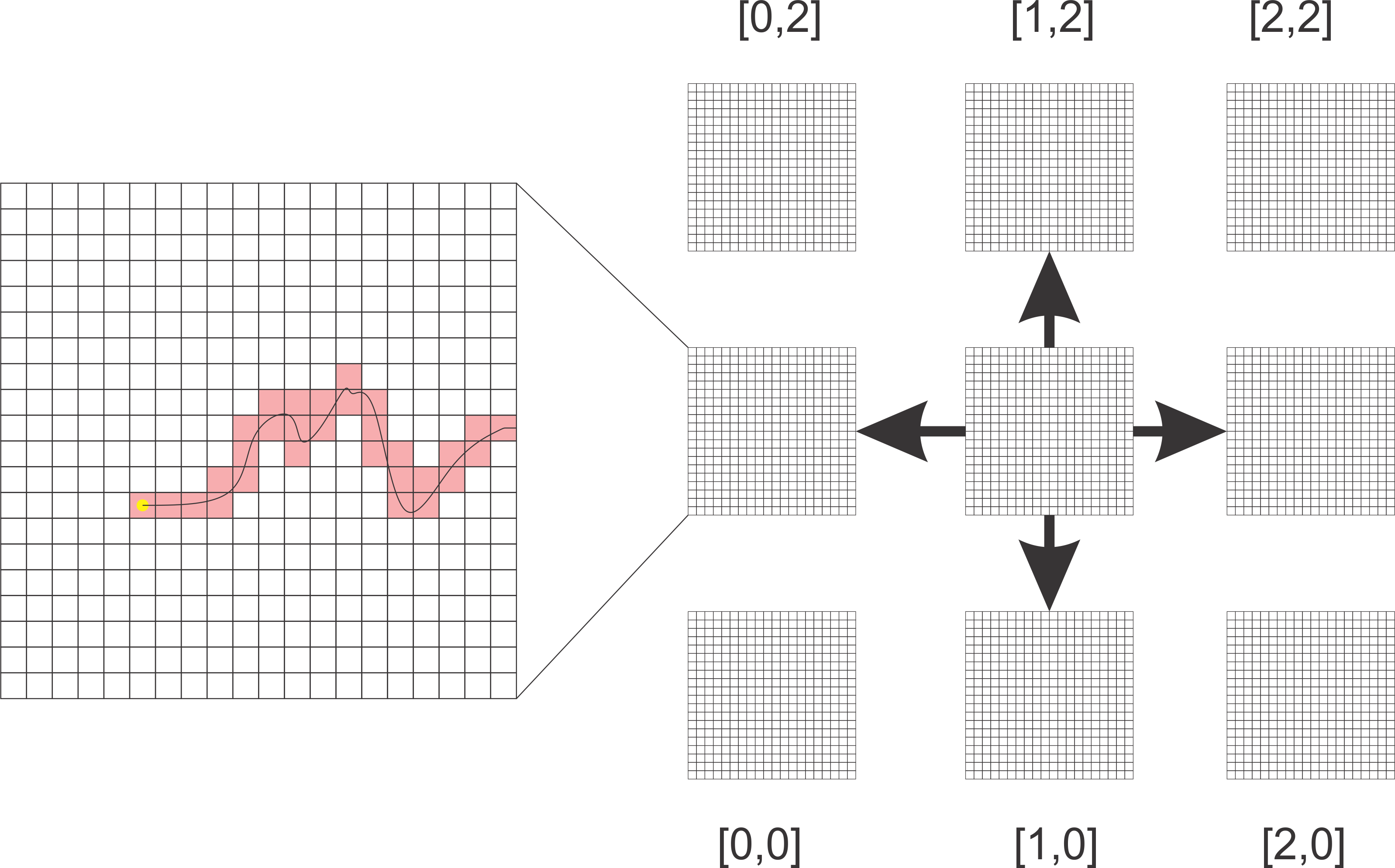
Start observing a person’s latitude (lat.) and longitude (lng.) data. For each locational datum, store an entry into a DAG with the vertex indexed by Mercator grid coordinates (x,y) in a virtual **spatial tree (R-tree)** stored in a database. The Mercator projection is the most suitable for this problem for two major reasons:

1. It is the projection used by Google Maps (needed for cross-reference & locational data lookup).
2. It preserves angles of navigation, thus producing a viable vector space for route storage in the graph.



**Figure 1: The Mercator Projection**

The Mercator projection will allow us to represent the entire surface of the earth using a 2:1 grid that is easy to decompose into data segments in a spatial tree. Each chunk consists of further chunks (or layers) down to a resolution of 100 feet by 100 feet. The following diagram conceptually defines the chunking algorithm. This data structure is called a **spatial tree**.

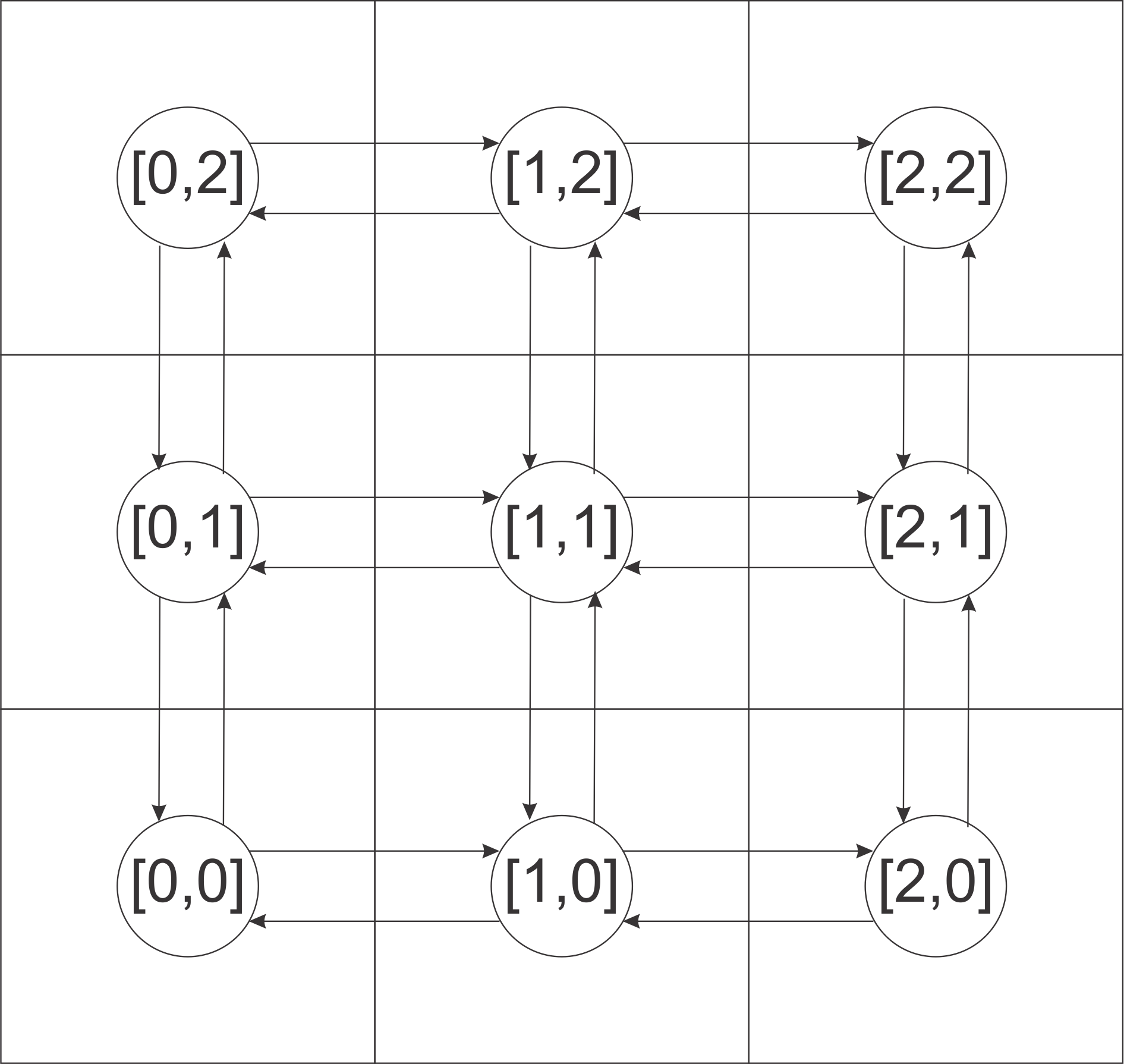


**Figure 2: Chunking and Graph Connectivity**

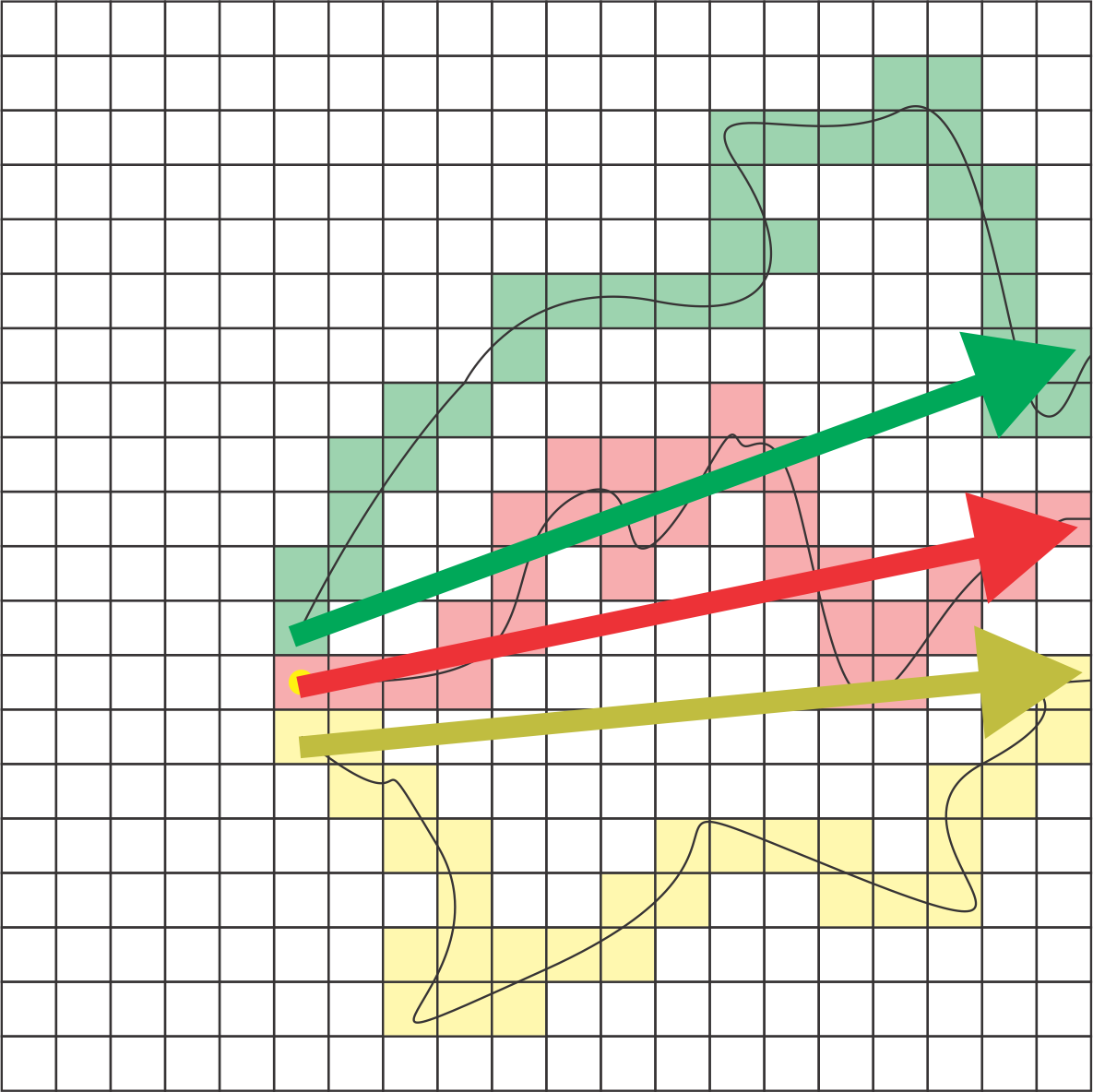
We break the space up into three layers. We loosely term these **Zones**, **Sectors**, and **Chunks**. This arrangement is also a hierarchy, so all the space in a zone is made up of sectors, and all the space in a sector is made up of chunks. This arrangement also allows us to design an extremely fast conversion from latitude and longitude directly to zone, sector, and chunk coordinates (a few CPU cycles). This is incredibly important because we will often query the spatial tree based on a vector and therefore want all sectors containing the vector.

Figure 2 demonstrates how a specific route will be captured in the raster. We use a raster in order to “digitize” the space and allow us to use integer coordinates in the algorithms used in the method. The raster forms an implicit 2D graph where coordinates on the four cardinal directions are connected, thus producing a graph where [1,1] is connected to [1,2],[2,1],[0,1],[1,0] but not [2,2].

This produces a graph like the one in Figure 3. Note that the graph is directed. This means that moving from Sector [1,1] to [2,1] is a different path to [2,1] to [1,1]. We will use the notation C[1,1]->C[2,1] to mean “moved from Chunk[1,1] to Chunk[2,1]” and so on. We use a short-hand alphabet to represent the possible paths; N, E, S, W i.e. the cardinal directions of the compass. In general, whether dealing with zones, sectors, or chunks, the logic is identical. This is because the spatial tree preserves the vector space – i.e. distances and directions are preserved. This is essential for rapid querying and eliminating bad matches.



**Figure 3: In- and Out-edges of Sectors in Spatial Tree**

****

**Figure 4: Angles and Distances are preserved**

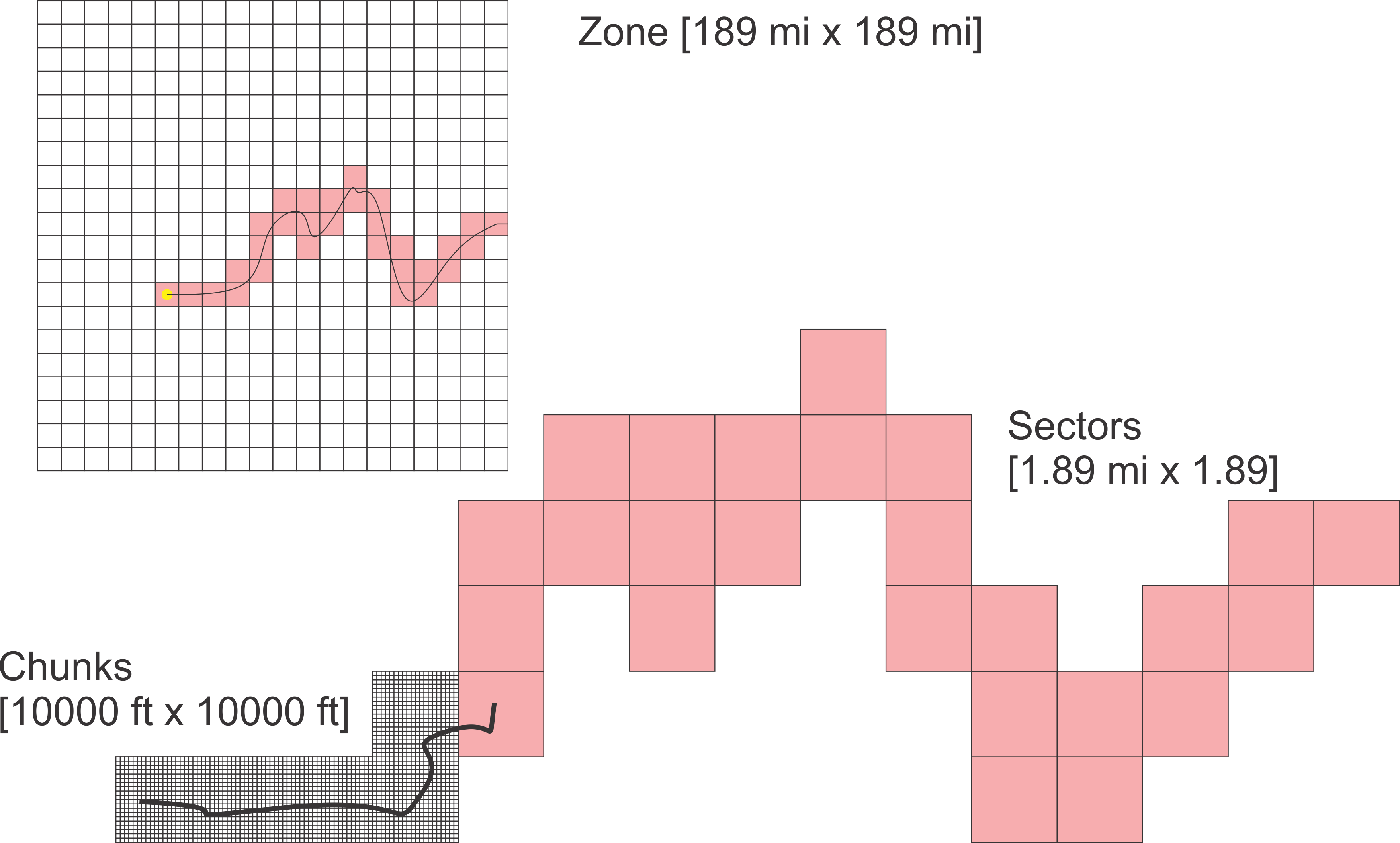
We define the smallest unit we can capture as an area of 100ft x 100ft. These squares are arranged in chunks of 100 x 100 units (representing an area of 10000 square feet) and so on.

We use the following terminology:

1. Level 3: **Chunk**: 100 x 100 units [10000 ft2 = 0.00036 mi2]
2. Level 2: **Sector**: 100 x 100 chunks [3.587 mi2]
3. Level 1: **Zone**: 100 x 100 sectors [35870 mi2]

There are therefore approximately 130 zones around the equator. The reason for adopting this chunking system is because of the following principle. A path between two points may be found by first searching for a path through the zones, say zone[1,1] to zone[2,1]. This means that the sectors between zone [1,1] and zone[2,1] must be searched to determine the path. The sectors under consideration are the sectors from [100,100] x [200,200].

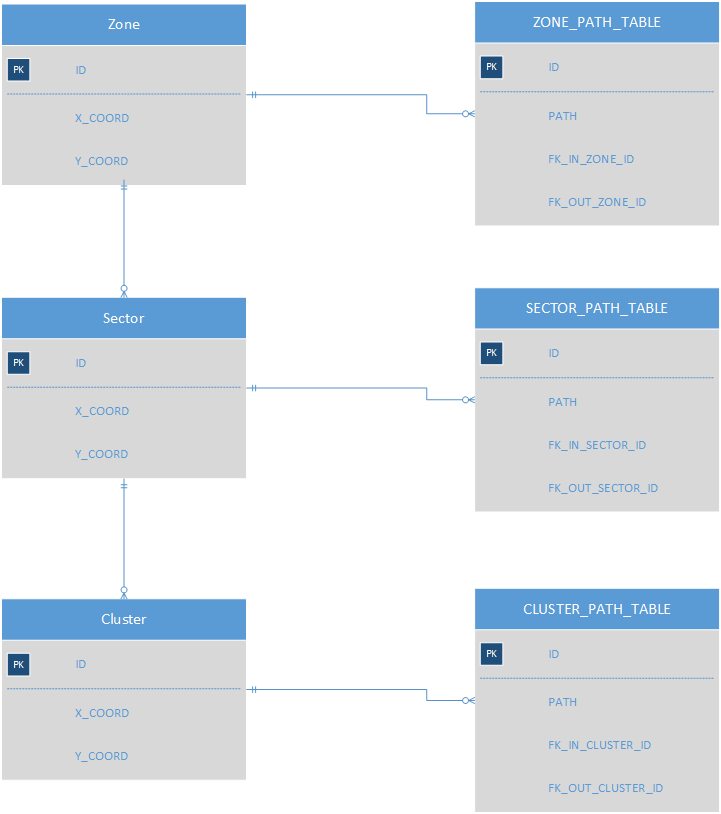
Furthermore, the path from zone[1,1] to zone[2,1] must therefore pass through the chunks [10000,10000] to [20000,20000]. The result is that we can iteratively decompose the pathfinding problem to smaller and smaller units in parallel – *a massive boon for speeding up matching paths*. The following diagram attempts to explain how sectors connect chunks together.



**Figure 5: Zones, Sectors, and Chunks**

This decomposition of both the learning and searching problem also allows one additional advantage. It is easy to define the data in a location-aware manner, thus allowing a simplification of the paging process and releasing of zones from memory. Furthermore, because the only communication between zones, sectors, or chunks is via the stitching area between two chunks (or sectors, or zones), the zones can be distributed throughout a cluster and processed simultaneously on multiple machines (e.g. one machine processes one sector, another processes a different sector).

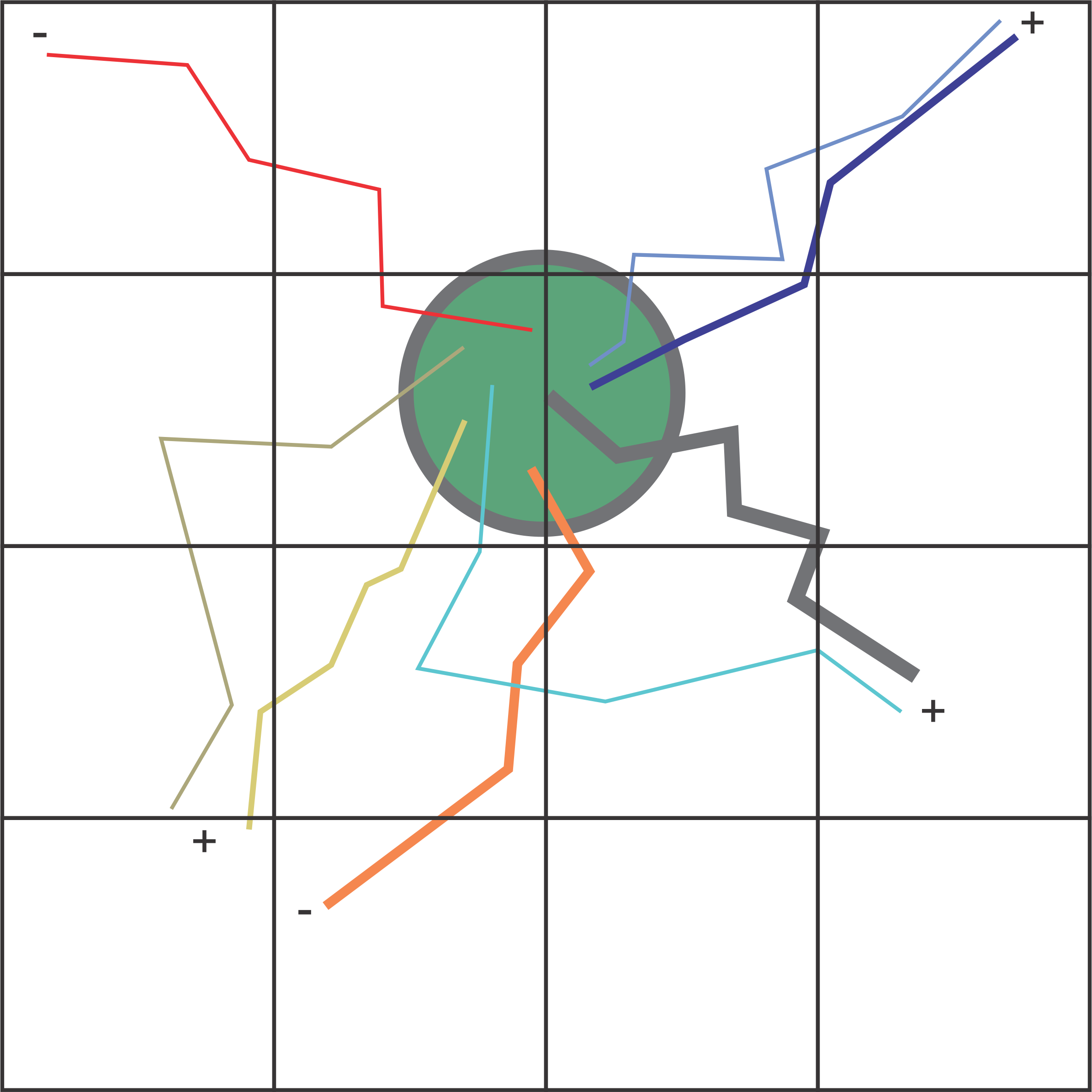
In order to query the system, we need to simply maintain a lookup table of all chunks with a path starting in them. This can be done in a database table and it maintains a pointer to Route instance – a data structure containing the directed arc data at the cluster level (i.e. in 100 ft2).



**Figure 6: Spatial Tree ERD**

Note that the FK\_IN\_CLUSTER\_ID in the above diagram is the originating cluster (or NULL if this is the start) and the FK\_OUT\_CLUSTER\_ID is the terminating cluster (or NULL if this is the end).

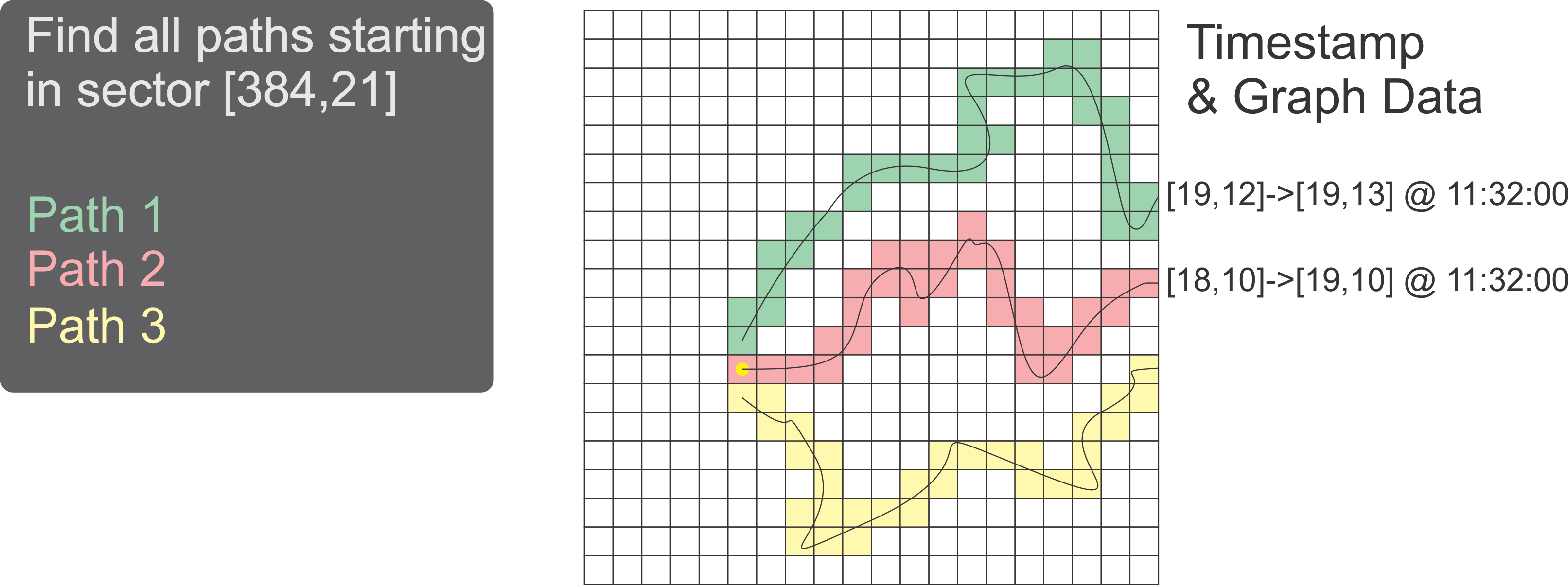
This allows us to query a sector for all paths starting there by aggregating all paths in all clusters for the sector. Therefore, the sector may also maintain an index of all paths as an optimization. This allows the query to rapidly drill down to the desired chunk and expand outwards using a raster disk-drawing algorithm (Bresenham’s circle + fill).



**Figure 7: Radius Query in Spatial Tree**

The disc is intersected with the spatial tree in order to determine which sectors to query for paths. This allows differential radii for queries and thereby a method for broadening or loosening the criteria for search. One the particular zone or sector has been found, the algorithm can start comparing individual paths to one another.

The following diagram shows a sample query where the system needs to examine all paths in an arbitrary sector to determine the best match.

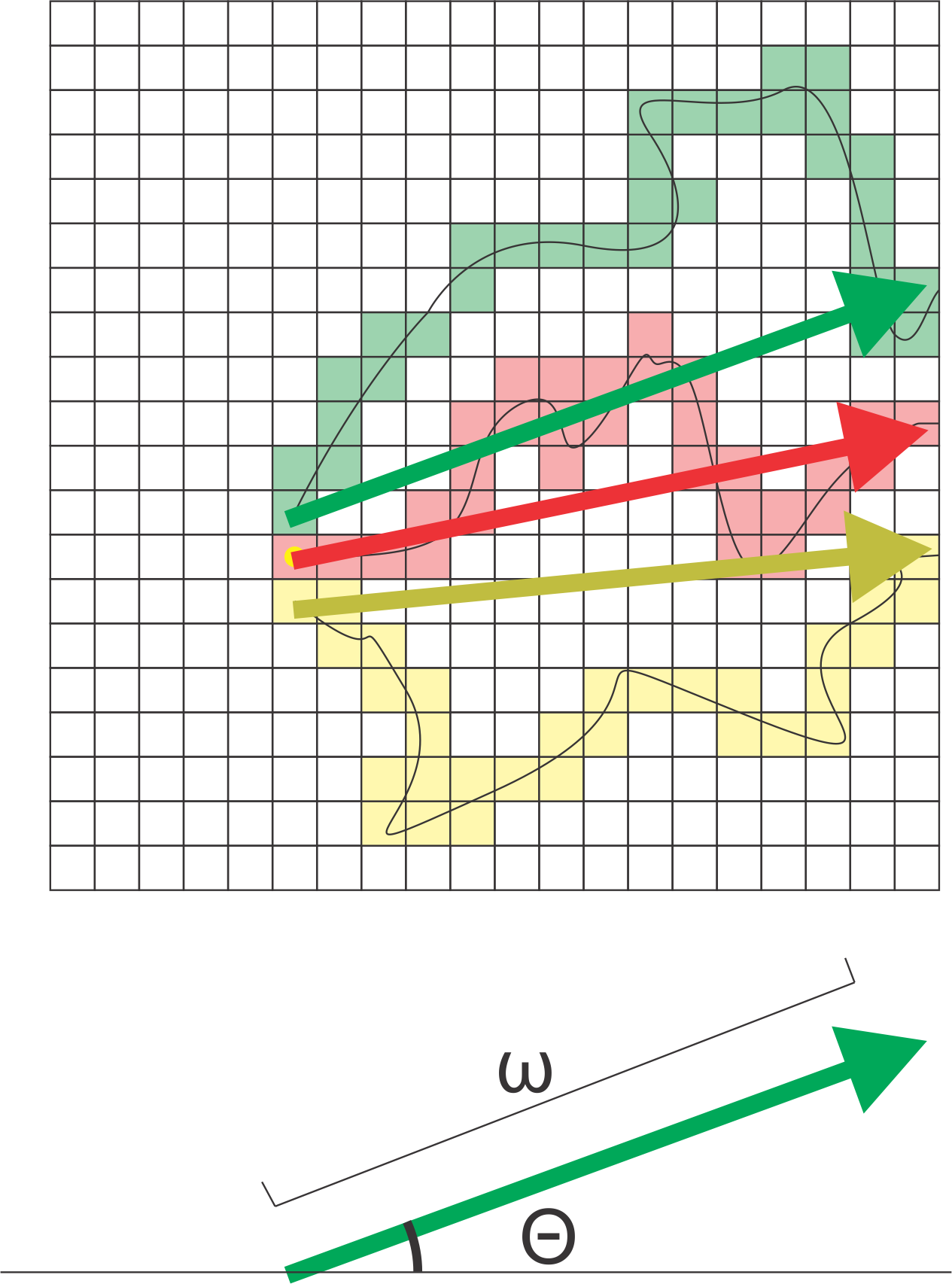


**Figure 8: Finding paths in a Zone/Sector/Cluster**

The query may now apply the following heuristic:

1. Compute the vector direction for each path in the sector (see figure 7).
2. For each path vector, compute the closest match on:
   1. Start Node
   2. End Node
   3. Direction
   4. Time
   5. Duration
3. Deviation
   1. Evaluate candidates by projecting graph transitions as fixed cost vectors onto target vector.
   2. Compute path deviation using cardinal direction vectors (explained below).

In the example below, the red path will first examine the yellow path for a match and then the green path for a match. The yellow path is the most similar in terms of direction and distance as per the following vectors.



**Figure 9: Vector Distance(omega) and Direction (theta)**

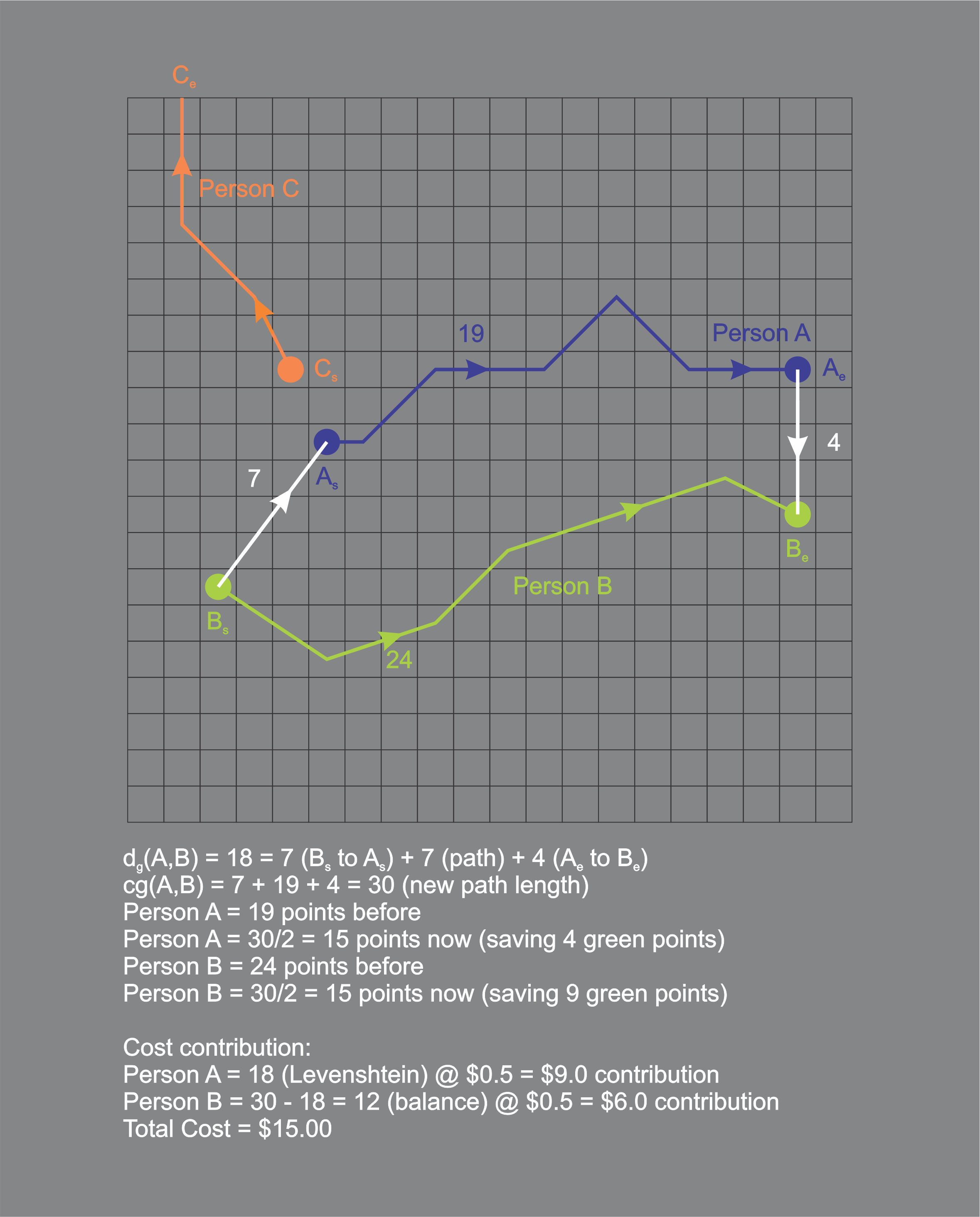
We can now use linear algebra (the Euclidean distance) as a short-hand for eliminating potential paths for green points matching. The vector gives us a rough path we can examine and the further we deviate from the vector, the greater the penalty for a match. This is the basis on which the Green Points method is initiated.

# An Example Case

We present a sample case that demonstrates the method in action and calculates the green points, cost contribution, and shows how the system chooses between one path and another.

We refer to figure 10 as the example case of three people, Person A, B, and C, with person A querying the system for a patch. Person A searches for all paths near her intended route within 10 minutes of her desired start time. The system returns figure 10 for analysis. We therefore have three different paths to evaluate and then we need to determine the cost proportions and green points.

The method first performs a rough analysis phase where the total displacement of each route is calculated. This means we calculate the “as-the-bird-flies” distance for each route. Person C is eliminated at this point because the heading is in a different direction and therefore will be highly unlikely to be a good match. This is primarily an heuristic and it the sensitivity can be configured.



**Figure 10: Green Points and Cost Contribution**

Next the system considers Person A vs Person B. In this case, the path distance is 18, very low and therefore a good candidate for route calculation. The system will compute the old A path (19 units long), the old B path (24 units long), and the smallest possible distance between the new path of 30 units long. The new path is 30 units long calculated by joining the B path to the A path and back to the B path. This is likely to be the most efficient route (because it is assumed that Person A has already selected an optimal route – however, it does not matter if s/he has not).

We will now see why the system will recommend that Person A join with Person B. Person A was spending 19 units to get to work. Person B was spending 24. By combining their route, the cost of the 30 unit trip can now be split in terms of carbon usage. Therefore, when allocating green points, it is recommended that the total carbon usage for the trip be split between the two parties equally. This does not have to be so however, the cost could also be allocated the same way as the financial contribution. In terms of the 50/50 split, Person A saves four green points but Person B saves 9 by sharing the path cost with Person A.

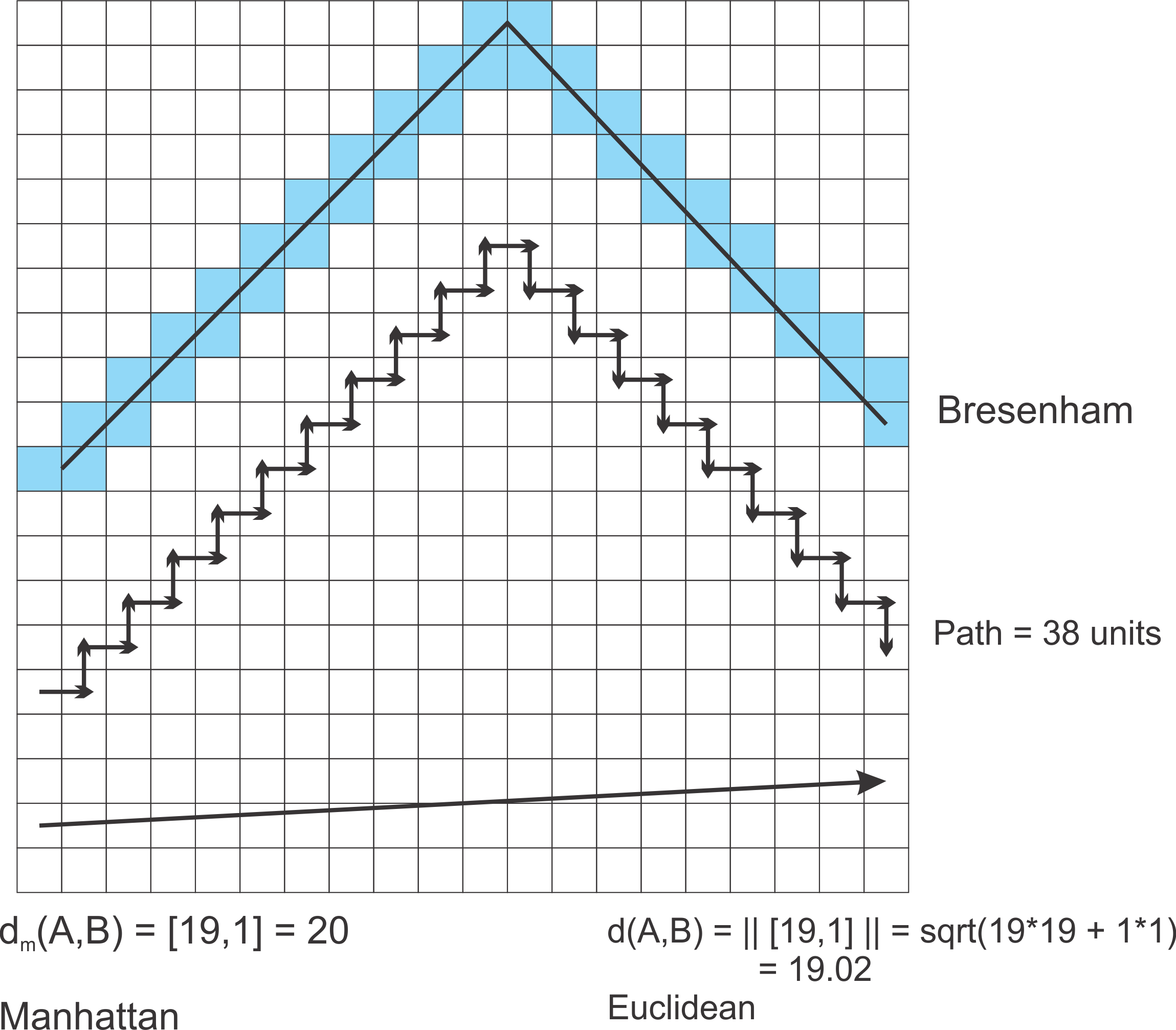
In order to determine the financial contribution we say, for argument sake, that each unit costs $0.50. Therefore, the old route A was costing $9.5 and the old route B was costing $12.00. In order to determine the cost distribution between Person A and Person B we make several observations. First, both parties are sharing some of the path because they were heading in the same direction; however, Person A requires some diversion from the path Person B would have followed and thereby increased the journey for Person B. It is therefore recommended that Person A contribute the Levenshtein distance between the old Path A and the old Path B. We can think of this cost as the amount of “diversion” required from Person B – a cost that Person A is responsible for. Note, however, that a more equitable contribution can be arrived at.

It is important to note that by establishing a regular metric on the concept of routes, the avenue opens to cost and apportion it. This is the goal of a good metric; to accurately provide a system for comparison of the objects under study.

Finally, note that Google Maps should determine the route between path A and path B. It is suggested that the system query Google Maps in a post-processing phase to calculate the true cost of the diversion on the actual network graph. The system will use the Manhattan distance as an approximation because it is likely to be a very good approximation; but emitting the connection points and calculating the equivalent cost in Google Maps should calculate the real cost of **only** the connection portions of the paths.

# A Mathematical Primer

GIS data is inherently mathematical. In order to accurately determine a comparison between competing paths, the system requires the concept of a **metric space**. A metric space is a mathematical structure implying that a non-zero distance between two points in a space can be computed and furthermore, this distance should behave according to the triangle inequality – i.e. the shortest distance between two points is a straight line. This is true of the metric chosen to represent path comparison, the **Manhattan Distance** metric (or taxicab distance).



**Figure 11: Manhattan vs Euclidean Metric**

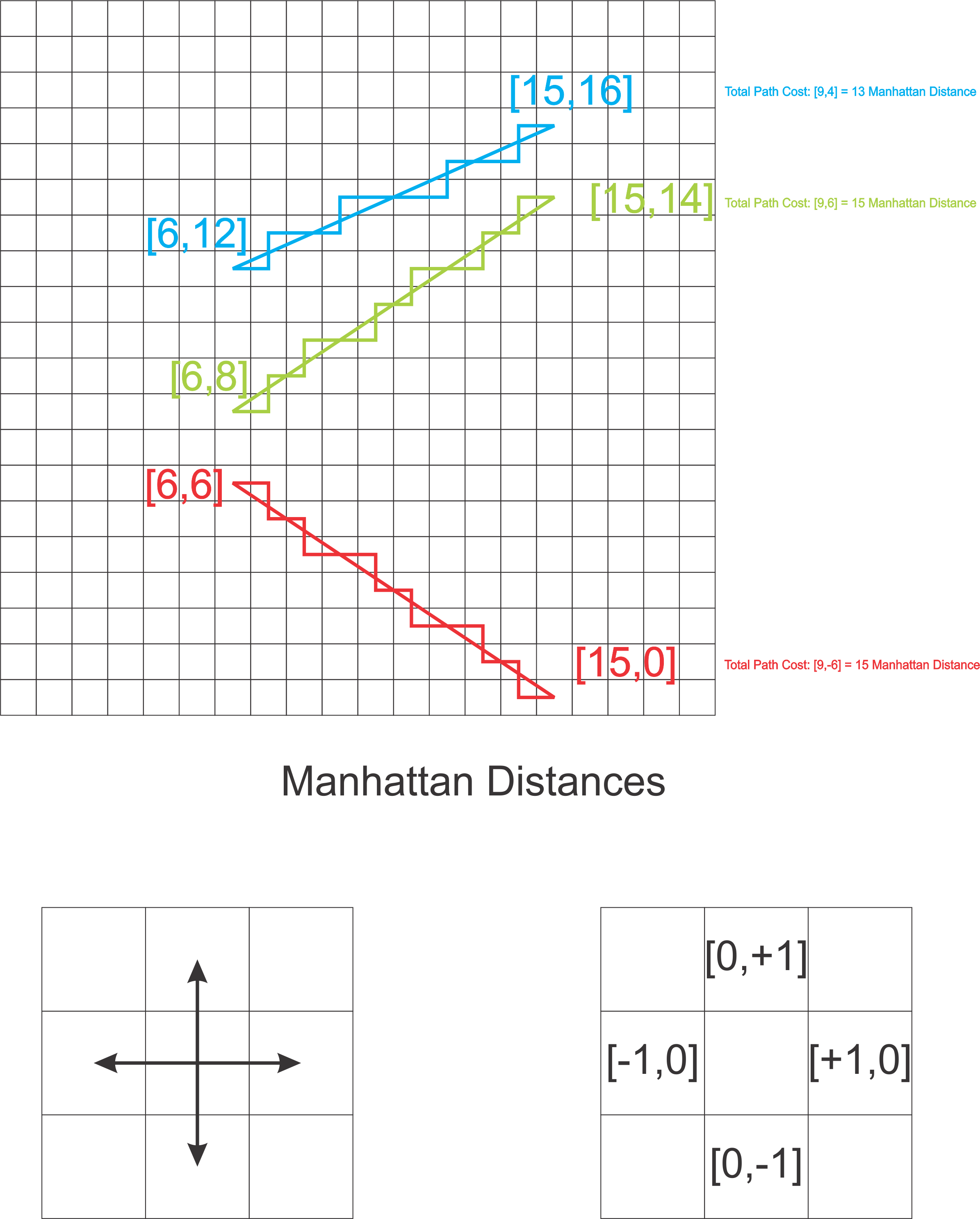
The Manhattan distance is chosen as the metric space for path comparison in the grid. There are several reasons for this. Firstly, because we have digitized the world map, we have a natural grid arrangement – the grid ordering (do not confuse the space with Euclidean space, the two are **not** the same) is required because it is an appropriate approximation of how distance works and it is very suitable for efficient path-matching and pathfinding algorithms by limiting the number of paths to be evaluated to a factor of four per node (i.e. 4^n where n is depth).

The Manhattan distance (metric) is defined thusly:

Macintosh HD:private:var:folders:f5:5_nslwv160vccmybcfw7qtyc0000gn:T:TemporaryItems:latex-image-1.png

This provides us a metric for calculating distance in the World Map. This is only the first element of computing a scoring algorithm for different paths however, because we also need a suitable method for comparing two paths in the world map and evaluating their similarity. Two important assumptions are required to develop a model for path comparison in the World Map.

1. A path can be represented by sequence of ordered directions (we can only travel in four directions [-1,0], [0,-1], [1,0], and [0,1]). Therefore, if we know the starting point, we only need to store the step-wise directions from four choices. Therefore, we can assign a letter to each direction (E,S,W,N respectively).
2. The displacement between two start-points and two end-points represents the added distance to be travelled when evaluating a path.



**Figure 12: N,E,S,W encoding of Manhattan Metric Space**

We make these two assumptions for a very important reason, the closer the two paths match in terms of the sequence of directions, the more parallel they are. Therefore, if two paths are identical in the sequence of steps required to travel from the start to the end, they are parallel and therefore identical in terms of direction and therefore zero cost (a perfect match).

We therefore use the concept of the **Levenshtein distance** to evaluate the two different paths once they have been selected for comparison. The Levenshtein distance allows for a cheap comparison between two paths in order to determine how many “edits” would need to be made to the tested path in order to transform it to the source path. This is crucial because it provides us a metric for comparing one path with another to evaluate the cost difference between the two. This is the crux of the green points algorithm.

The Levenshtein distance metric is a measure of the distance between two input strings. The strings are evaluated by comparing whether each point in the string is a match or whether a character should be inserted, deleted, or replaced to produce a better match. It is a rigorous definition of the distance between two strings but also a highly optimal measure of the directions taken in N,E,S, or W coordinates. This allows a highly optimized algorithm to determine the deviation of two paths in a graph by encoding the directions of the path as letters of an alphabet. The Levenshtein distance metric is therefore the heart of the UBL method.

# Green Points Algorithm

Consider trying to determine whether the paths A and B are a good match. The goal of the green points algorithm is to determine the best match between a candidate path and a set of roughly matching paths. The smaller the value produced by the green points metric (**dg**), the better the match. We therefore seek to minimize the green points score when evaluating paths. Taking the aggregate direction between the source and target paths eliminates initial paths if they are heading in a very different direction (in Euclidean space):

Macintosh HD:private:var:folders:f5:5_nslwv160vccmybcfw7qtyc0000gn:T:TemporaryItems:latex-image-1.png

Macintosh HD:private:var:folders:f5:5_nslwv160vccmybcfw7qtyc0000gn:T:TemporaryItems:latex-image-1.png

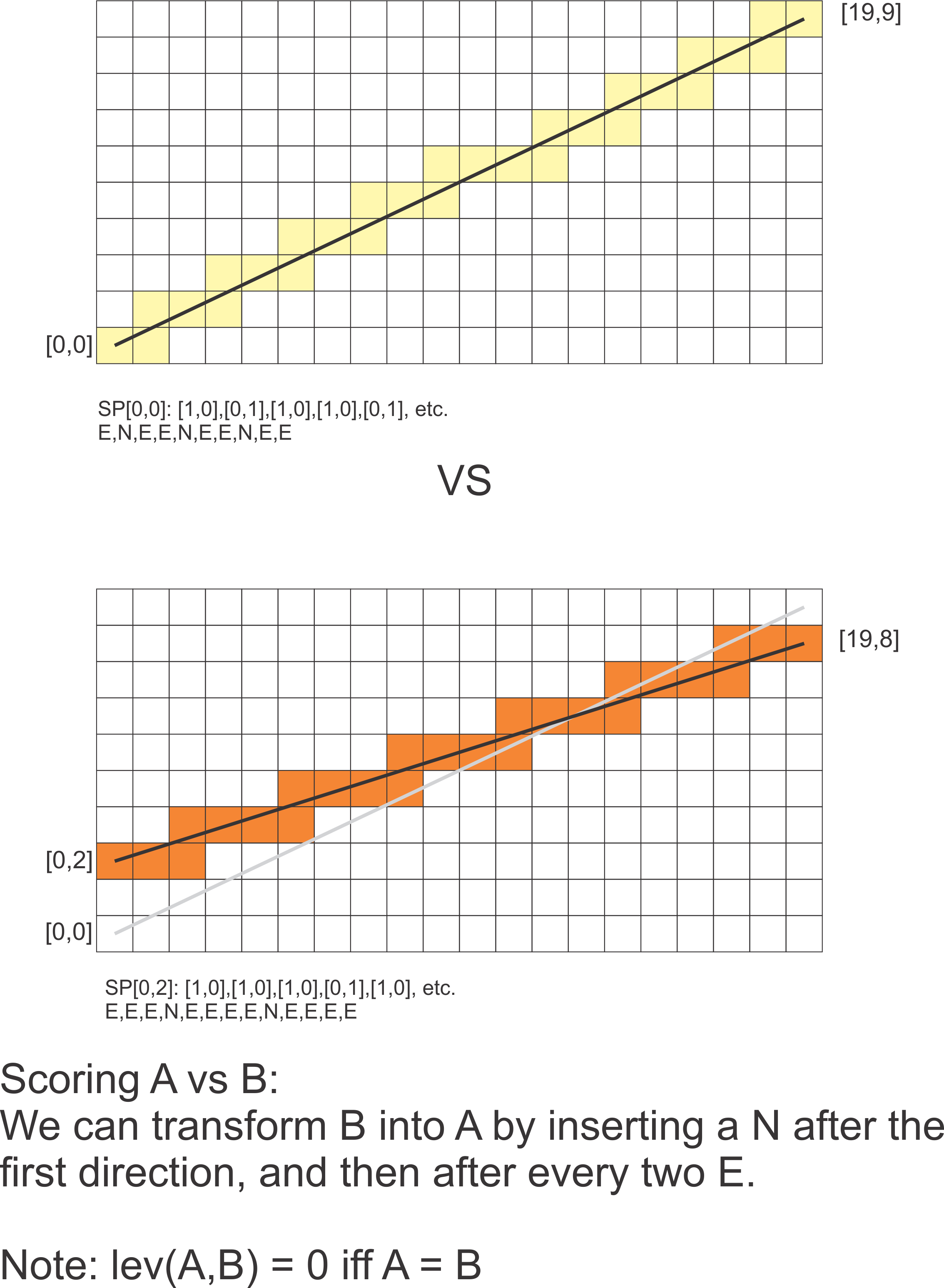
Thus, if the vectors are differing greatly and not heading in generally the same direction, the algorithm can quickly eliminate the path from further checking (note we are checking an area so we already know the paths are near each other). This allows us to quickly eliminate vectors that are going in the wrong direction and therefore guaranteed to result in a high green points cost. The further the path deviates from the source path, the higher the green points cost.

We observe that there are only four possible directions in our grid. We can head north, east, south, or west. This allows us to create an alphabet for encoding paths, requiring only 2 bits per transition! Thus, a path is encoded by storing the start point and a path “string”. This is crucially imporant because of the Levenshtein distance metric – allowing us to directly compare two paths for deviating costs. This is the method by which green points are computed.

Once two paths have been selected for a possible match, we execute the green points metric as follows:

1. Compute the distance between the start of the source path (As) and the start of the comparison path (Bs).
2. Compute the Levenshtein distance between the two strings (the whole distance could be computed by using Levenshtein but it will be more efficient to measure the Manhattan distance (a Bresenham raster line) between the starts and ends rather than building a larger matrix for Levenshtein comparison).
3. Compute the distance between the end points of the two paths.
4. Add all three values together and this is the total cost difference between V and W.

The algorithm can thus be defined as performing the following operation:

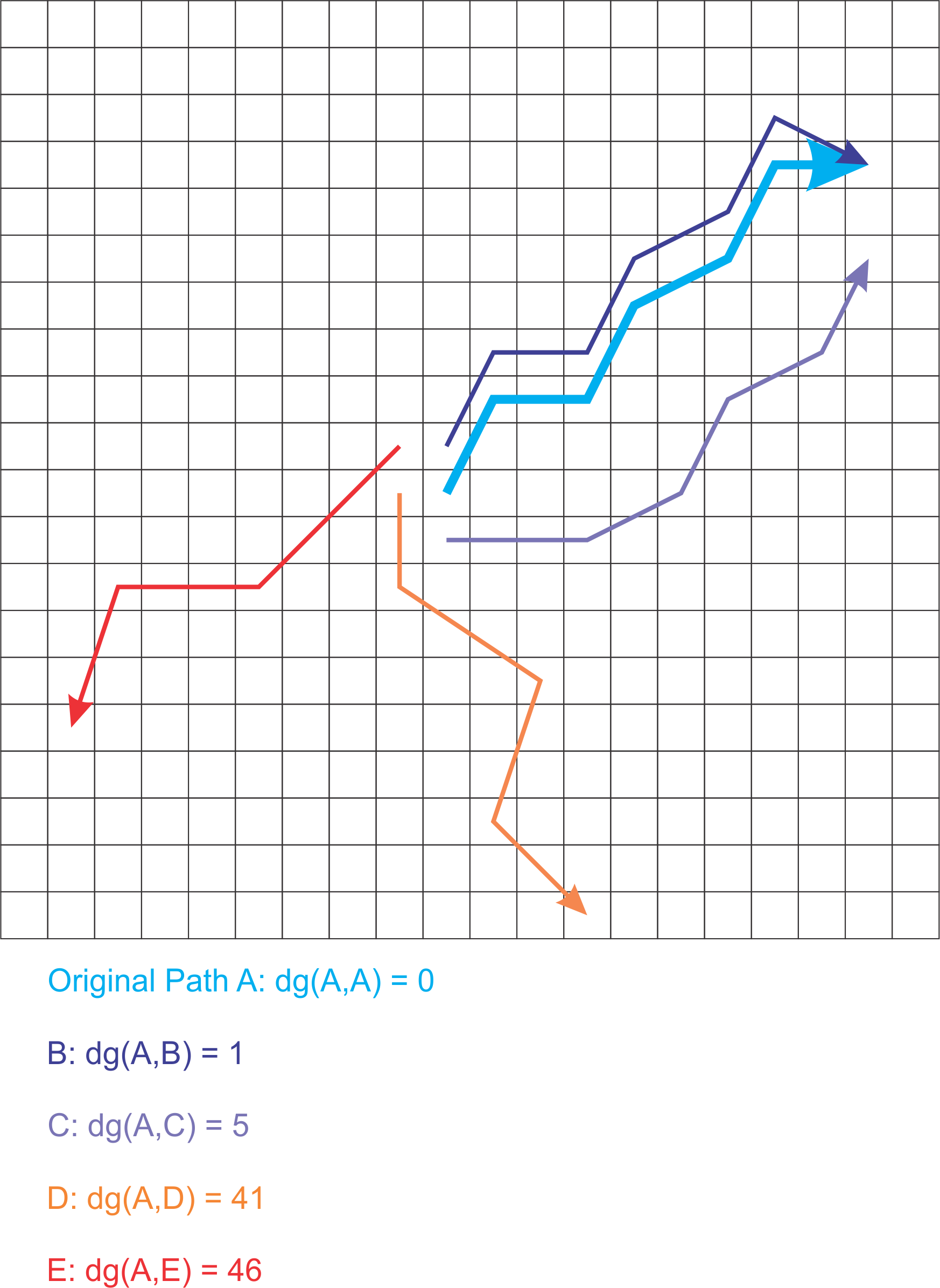


**Figure 13: Levenshtein Metric**

## In Practice

The following diagram demonstrates how the green points metric works in as a distance metric. The following diagram shows several paths compared to the initial A path. The blue paths match the original path quite closely, in particular the dark blue path which is nearly parallel and ends at the same destination. In this case, if the light blue path were paired with the dark blue path, the total “additional” cost would be only one unit.

In the case of the purple path, the additional cost would be 5. This is due to the fact that the path is both displaced and deviates from the source path.



**Figure 14: Green Points Scoring**

Effectively, the green points metric calculates the proportional cost of sharing the path and the added cost of connecting the target path with the origin path. This means that in order for the person driving the dark blue path to match the light blue path, a single unit is required and therefore the best, cheapest match. The additional unit required to connect the two paths should be paid by the searcher in full whereas the remainder of the path can be shared between them. This is the goal of the green points algorithm.

The purple path is not as good a match because of the increased distance between the start and end points, and because the purple path is more costly overall. It therefore costs more “green points”. In the case of the orange and red paths, the cost is outrageously high. However, both the red and orange paths would have been eliminated prior to the computation of the Levenshtein distance.

## What about Time?

The final aspect required for matching paths is the temporal matching between paths. It’s all good and well that we can match the paths in xy coordinates, however for the system to work properly, we will need to match the proximity of the times of the paths as well.

Fortunately, all the concepts introduced so far extend into three dimensions. One can visualize the time dimension has a height field on top of the map wherein the path increases in height through the passage of time. The exact same concepts allow distance comparison to time by simply extending our vector space to be (x,y,t). Furthermore, a Levenshtein distance can be computed to also factor time into the green point calculation. Thus, if the target path starts after the source path, the distance will be greater and therefore cost more to ensure that the target match is remunerated for the additional time.

It is also important to note that the time dimension is strictly monotonic, i.e. it can only increase. This alters some of the calculations somewhat, but can be thought of as follows: if a path is stationary, it does not move in the x,y coordinates but always increases in the time dimension; it therefore means that the x,y coordinate is repeated in the next “time cell” when computing the distance. It can be difficult to visualize if not familiar with vector spaces; but it is rigorously defined.

## Algorithm Parameters

The algorithm can be manipulated using several parameters:

* Path Proximity – prefer closely matching paths
* Temporal Proximity – prefer closely matching time
* Shortest Path – prefer shorter paths over longer
* Slower Path – prefer slower paths over more expensive paths

Each of these queries can be answered by a relative manipulation of the model parameters. The distance metric thereby provides a solid mathematical basis for comparing paths to each other.

## Calculating Costs

The algorithm will use the following functions:

Macintosh HD:private:var:folders:f5:5_nslwv160vccmybcfw7qtyc0000gn:T:TemporaryItems:latex-image-1.png

Macintosh HD:private:var:folders:f5:5_nslwv160vccmybcfw7qtyc0000gn:T:TemporaryItems:latex-image-1.png

Macintosh HD:private:var:folders:f5:5_nslwv160vccmybcfw7qtyc0000gn:T:TemporaryItems:latex-image-1.png

Macintosh HD:private:var:folders:f5:5_nslwv160vccmybcfw7qtyc0000gn:T:TemporaryItems:latex-image-1.png

And therefore, the following formulas apply

Macintosh HD:private:var:folders:f5:5_nslwv160vccmybcfw7qtyc0000gn:T:TemporaryItems:latex-image-1.png

Macintosh HD:private:var:folders:f5:5_nslwv160vccmybcfw7qtyc0000gn:T:TemporaryItems:latex-image-1.png

Macintosh HD:private:var:folders:f5:5_nslwv160vccmybcfw7qtyc0000gn:T:TemporaryItems:latex-image-1.png

Macintosh HD:private:var:folders:f5:5_nslwv160vccmybcfw7qtyc0000gn:T:TemporaryItems:latex-image-1.png

# Input/Output

In learning mode, the algorithm expects the following data:

PERSON (ALISON1234)

LOCATION (LAT: phi, LNG: lambda)

TIMESTAMP (2014-11-11T15:34:04.123456)

This is the only data required for the system to learn the paths taken by the person. Several heuristics will be used in order to determine when a “Route” occurs. The following heuristics are proposed:

1. A route will begin after 30 minutes in a stationary position. A stationary position should be defined as some radius (it may be larger than 100ft) under consideration. The person will start a new route when leaving a radius after a period of 30 minutes.
2. The route will be recorded by monitoring all timestamps in the chunk as traversals in the chunk. **The data must be interpolated**. This means that if the recording stops at 10:10 in chunk index [10,10] and starts at 10:30 in [10,15]; the person has moved from [10,10] to [10,15] through [10,11], [10,12], [10,13], [10,14] and finally to [10,15] at a constant rate and each hop **must** be recorded. This is required by our string-representation of traversal directions.
3. If a person is stationary for over 30 minutes, a route is determined to end. This will capture events such as “going to the shop” or “going to the doctor” or “going to work”.
4. A route will be created if the person travels a path and then returns along a “similar” (to be quantified) path. This allows the system to learn events such as “drop kids off at school”, “fetch a package from post-office”, and so on.
5. If a user is in the same location for under 30 minutes, the data will be compressed by removing all additional recordings in the cell between the first and the last. The person is assumed to move at an constant speed throughout the cell. Transitions are calculated by midpoint.

# Mathematical Definitions

We define three conceptual structures:

* World Space [0,0] -> [1314825, 656304].
* A Quadtree spatial index on the world-space containing Route Start and Termination Nodes.
* World Graph – a 2D rectangular, **connected, planar graph** (conceptual).

In order to determine the vector space involved:

(1 mile = 5280 feet)

Longitude = 24,902 miles = 131482560 feet.

Therefore the **x** domain = [0,1314825] in units of 100.000045633 feet.

Latitude = 24,860 miles \* ½ = 12,430 miles (pole to pole) = 65630400 feet.

Therefore the **y** domain = [0,656304] in units of 100 feet exactly.

We therefore round to 100 feet and accept the approximation error rate of:

Macintosh HD:private:var:folders:f5:5_nslwv160vccmybcfw7qtyc0000gn:T:TemporaryItems:latex-image-1.png

Which is certainly well-within necessary tolerance. We may therefore assume that all paths are drawn on a 1314825 x 656304 bitmap and we will express the location data in this coordinate system rather than latitude and longitude. This allows the data structures to be sparsely allocation (using a quadtree for initial querying) and therefore no graph will be allocated to store those parts of the world where no data is present.

Note: Altitude is ignored for the purposes of this algorithm. Therefore the algorithm considers only lambda and phi in the spherical space – an entirely suitable choice for a top-down grid-based view of pathfinding.

## Converting Longitude & Latitude to Mercator

The system requires a linear vector space in which to operate. The Mercator projection is perfect for this purpose and is used in Google Maps.

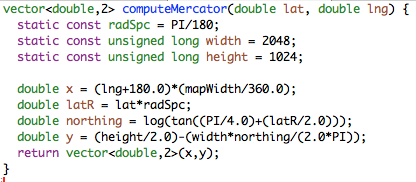
Mercator projection used internally

The latitude and longitude will be digitized/rasterized.

Conversion thus:

Macintosh HD:private:var:folders:f5:5_nslwv160vccmybcfw7qtyc0000gn:T:TemporaryItems:latex-image-1.png

where phi is latitude and lambda is longitude. The following code will convert from latitude and longitude into xy coordinates.



# Software Design

The software will require a multithreaded implementation in order to distribute computational load. It is doubtful that the algorithm will perform adequately in a single-threaded environment. It would be helpful to distribute the map in a cluster and divide the clusters into collocated zones. The advantage of doing this is that all searches within a zone will use data on a single machine, thus allowing shared memory implementations to reduce memory pressure.

It is important to note that the grid is conceptual and so is the graph. In order to find the correct zone we simply need to transform the latitude and longitude into the appropriate x and y coordinates and find the relevant zone. This can be done in a few CPU cycles and is a consequence of the Mercator projection.

Once we have found the zone/sector/chunk in question, we need to either process a new data stream (learning mode) or query the model for the closest matching path (query mode). We will discuss the design requirements of each in turn.

## Learning Mode

Learning mode requires an input data stream. This can happen either periodically or in bulk. The data stream will be analysed and converted from latitude and longitude into (x,y) coordinates in the Mercator grid. A path is recorded by using the heuristics defined above. Some experimentation and real-world observation of data will be required in order to tune the model to the realities of GPS reception and data recording.

The learning mode is essential for producing good query data. We require a method that averages input data and therefore it is proposed that an **Epsilon** value be introduced that determines repetitive paths.

Learning mode requires fast access to the user data but not the world map. Where possible, the user data should be cached in an LRU cache so that active users are kept in memory. Data in learning mode should be stored associated to the user and using xy grid coordinates and a sequence of timestamps. The data will **not** be interpolated yet. The learning mode will simply collect data for the analysis phase to be completed in 24 hour blocks.

The analysis phase will begin by examining the data and translating the feed into physical coordinates in the graph. This means that the analysis phase will interpolate (or fill in the blanks) in the data and translate the possibly broken data stream into start coordinates and a string encoding the exact directions taken from the start. The analysis phase will essentially construct a **polyline** and interpolate between coordinates missing in the datastream.

The algorithm will then break the data apart using the heuristics defined above; namely:

1. Repetitive route following.
2. 30 minute inactivity (or longer).
3. User classification of a path.

The system will average the data for each route in an attempt to classify those routes that are most frequently travelled. A numerical value will be assigned to the route indicating the “strength” or “frequency” of the route. This will allow the system to eventually arrange the data in order of most commonly used routes. The Levenshtein distance could naturally be used as a metric for determining route similarity.

The system will finally write the start and end points of all defined routes into the world graph (chunks, sectors, and zones). The data stream will be recorded as a sequence of “movement characters” along with a sequence of time recordings in seconds since the start of the route. It is preferred that we store the differential data rather than the full xy coordinates or the timestamp. This will drastically reduce the amount of space required to store the path data. This is a trade-off with runtime as the processing of this data will naturally be slower. However, considering that I/O is vastly slower than a few additions, it is considered to be an appropriate trade-off.

## Query Mode

Selecting a candidate path and asking the system to find all closely matching routes in the system will initiate query mode. The system examines the starting point of the candidate path and queries a quadtree with a radius and starting point of the candidate path. The disc is intersected with the quads in the quadtree and all path starting points within the radius are retrieved for examination.

The quadtree can be located on a single machine and queried across the network. It makes sense to locate the quadtree on a single machine because a fast, array-optimised implementation can be used that ensures the data in a block is located near other data in the same block. This results in a massive boost in querying the quadtree for point data.

Once the candidate paths have been returned, each path needs to be hierarchically evaluated in a series of tests between the search and candidate paths. First, the zone transitions are examined and computed. Any paths ending in different zones can be eliminated as well as any paths that head in a different direction.

Matching zones will be quite easy to find. Next, we perform the same computations for the sectors and clusters. Note that the Levenshtein distance metric can not be distributed. Therefore, each comparison will need to run in a single thread. Therefore a multithreaded path comparator needs to be constructed for submission of distance calculations.

Once the distance calculations have been gathered and ordered, the top X number of matching routes (in order of Green Points) may be selected for return. The search should have parameters for relaxing either the time factor or the displacement factor. This will allow the user to be more flexible in terms of either the exact route or the time of day the route is taken. For people, probably the former, for packages, probably the latter.

## Data storage

It has been proposed that Cassandra be used as a data store. The data store presents no real limitation for the system and therefore any RDBMS-based database will be acceptable.

# Development and Testing

The development of the UBL method is fairly large. Furthermore, because the system is a *de novo*implementation of a path-finding system, the system will require very thorough testing. It is proposed that the development proceed as follows:

1. Implement the world spaces and chunking methods, build appropriate conversions to and from latitude and longitude data and correlate to Google Maps. Ensure space is consistent and overlaps closely to Google Maps.
2. Develop a set of random walk agents for data production. These agents should move around the graph and record their location as realistically as possible. They should be programmed to produce data like a real human would and it is recommended that the agents be designed to simulate all types of behavior we wish to record.
3. Develop the learning and pattern recognition methods based on agent data. Start capturing real-world data from actual users.
4. Implement a distributed system for performing path matching queries. As parameters, all that is required is the starting point of each path, and the sequence of directions and temporal differentials. Each path can be evaluated separately and therefore no shared memory problems arise here.

It is expected that the development will take approximately 3 months full time development.

## Akka vs Netty

There has been some discussion about the choice of Akka vs Netty. Akka is simply a high-level framework for distributing work easily between worker threads or machines in a cluster. It simplifies the development process by providing a DSL for data interchange and message passing. The actor model is a simple and easily understood model of computation and requires less effort to use than Netty.

In terms of implementation of the UBL method, there is no expected benefit to using Netty. The UBL method has been specifically developed to avoid massive data passing and generally no data needs to be shared. The program consists of at most three or four different actors responsible for different parts of the algorithm. There is therefore no expected benefit to using Netty in the algorithm. It is expected that if low-level threading is used instead, at least an additional week of development time will be required.

# Risks and Assumptions

There are several assumptions and issues in the method. These can be broken down into model risks and development risks.

Firstly, the model risks lie in the projective spaces used to transform the world map into a rectangular space. The projection will result in distortion (especially as the user moves further north or south). However, the Mercator project does preserve relative distances within a manifold and these Euclidean manifold exist in the projection. Furthermore, Google maps also uses this projection and therefore we can assume that the project is good enough given the distances under consideration.

The map cannot be regularly tessellated. Therefore, the map must be stitched at the 180 degree longitude. This requires special “fiddling” with the distance metrics in order to wrap the map around. It is fortunate that relevant longitude is mostly ocean and covers very little land-mass (Russia).

Finally, there is an assumption that the paths are short enough to be adequately computed using Levenshtein distance, but these methods have been successfully applied in genetic sequence matching and therefore it is unlikely that the paths in UBL will exceed the capabilities to compare them. Note Levenshtein is itself a graph algorithm and requires computation of an adjacency matrix.

# Conclusion

The UBL method is a viable set of algorithms for computation, comparison, and billing of pathway data in a spatiotemporal space. The method will allow an accurate costing of “Green Points” and facilitate the equitable exchange of funds for saving energy. The contributors are each likely to benefit by sharing the cost of the travel and reducing their carbon footprints in the process.

The system above is a suitable solution to the problem of accurately defining the concept of “Green Points”. Furthermore, the system is capable of performing intensive graph operations at a fairly low cost through repeated hierarchical decomposition. While the system might appear to be quite complex, the mathematics behind the entire system is actually quite basic and well-defined.