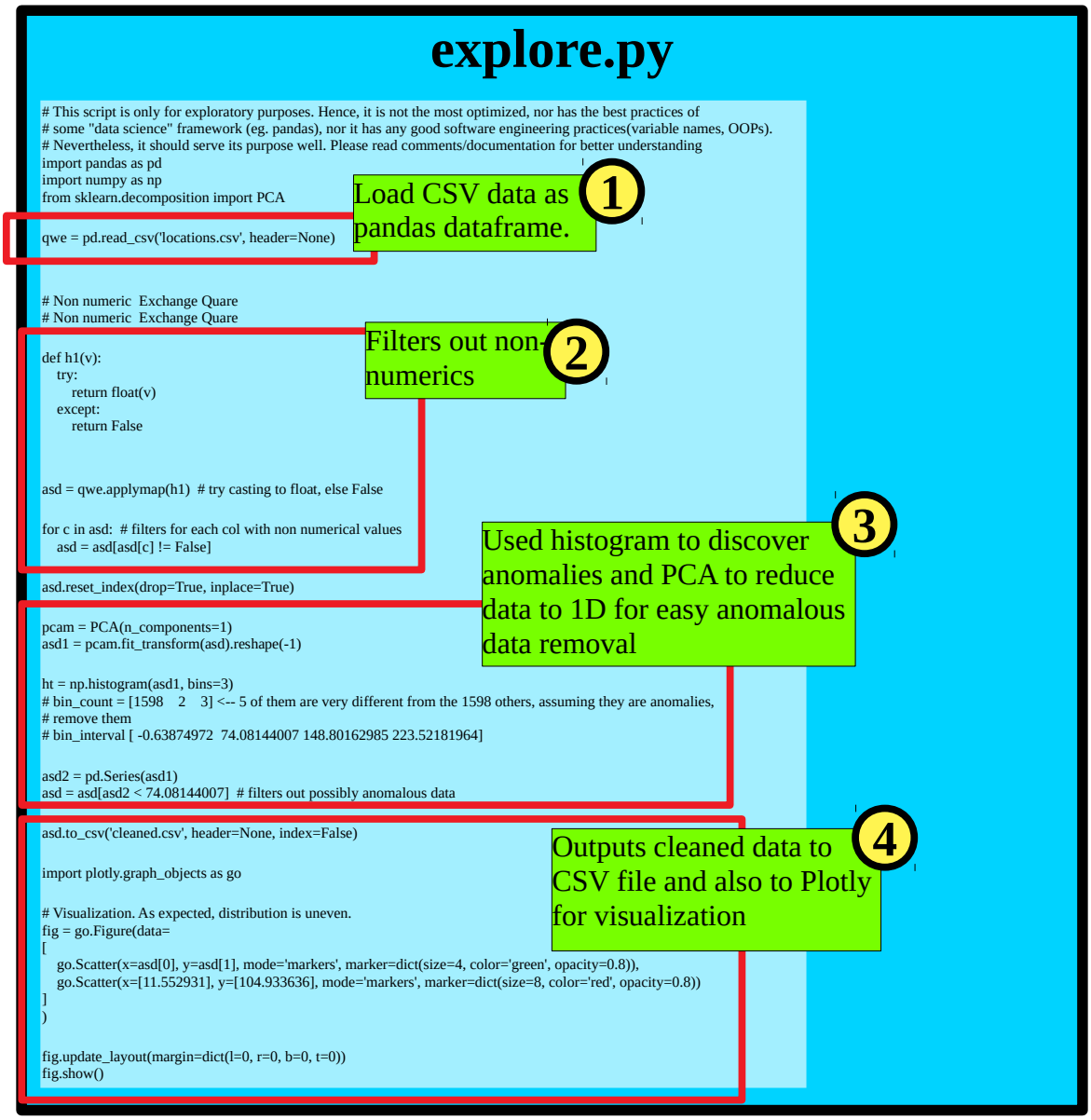
Efficient path solution for N workers

Table of Contents

1. Data cleaning

The data cleaning workflow is depicted in the following diagram. This workflow is specific to the data given in the assignment and may not work for other cases.



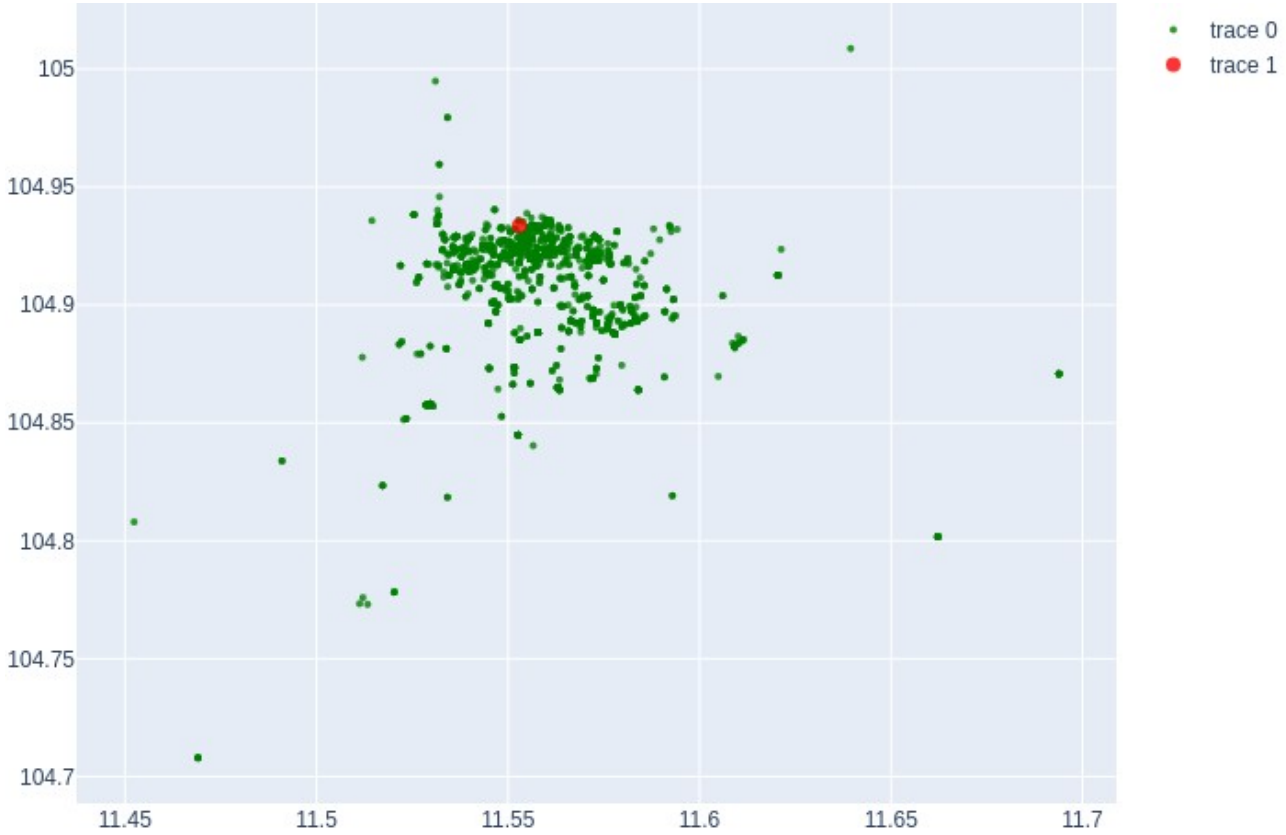
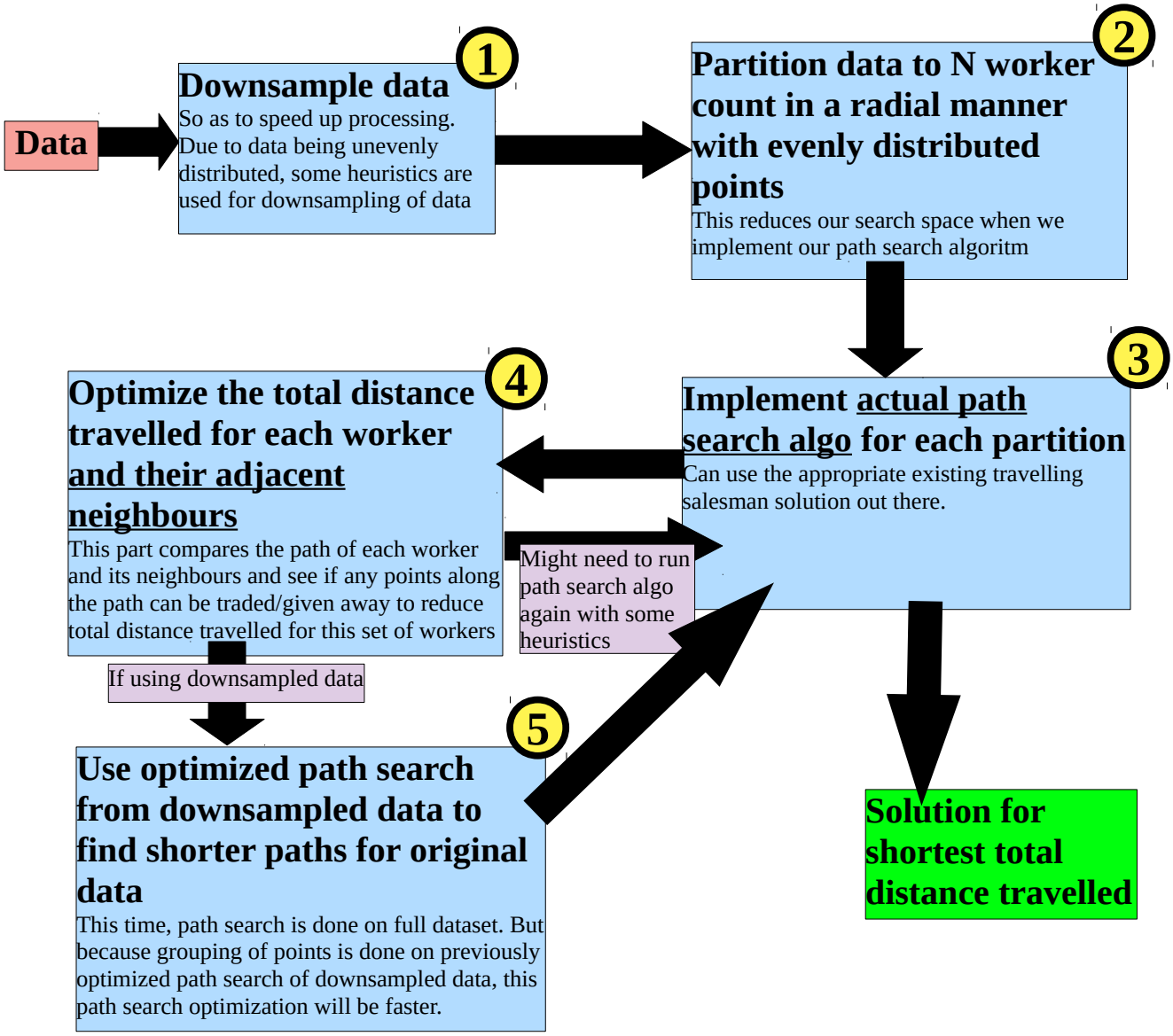


Fig. 1: Visualization of cleaned data

2. Path finding solution

We assume the optimization goal is to ensure total distance travelled by all workers is as short as possible (not the same as evenly distance travelled)

For the purpose of sheer processing speed and multithreading, Golang is used for this part here. The workflow is as follows:



2.1. Downsample data

To speed up solution search, downsampling is done. This downsamping method seeks to reduce data points that are similar, while keeping dissimilar data points, hence the downsampled data still maintain a fairly accurate representation of original data

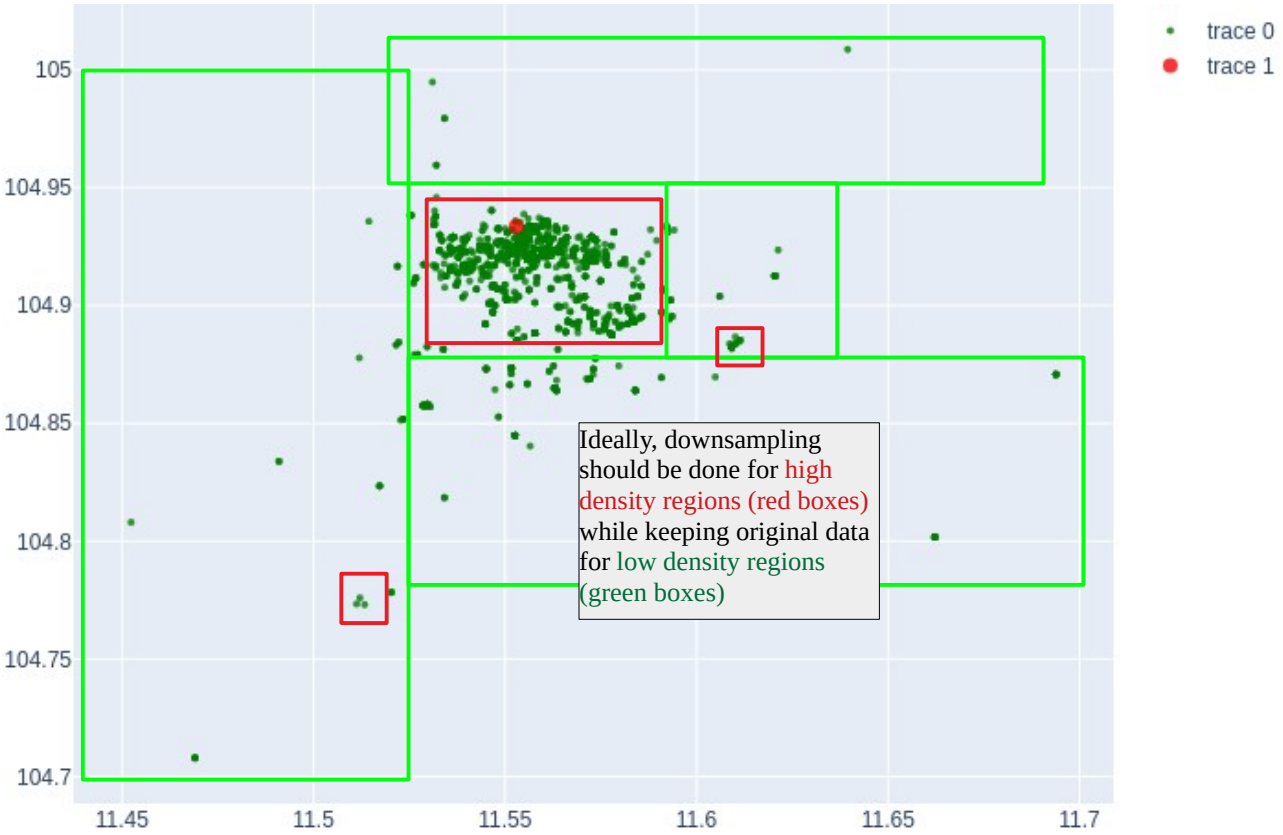


Fig. 2: Possible regions for downsampling

Actual algorithm

In the actual algorithm, data's first principle components and their mean/standard deviation is used for filtering out similar data.

pathfinding.go/downsampleSimilar

Actual values used:
SelectionRate=0.2
SelectionStandardDeviation=2

```
func downsampleSimilar(d [][]float64, selectionRate float32, selectionStandardDeviation float64) [][]float64 {
        rand.Seed(time.Now().UnixNano())
        dest := floats2DToDense(d)
        rlen, _ := dest.Dims()
        pcModel := PrincipalComponentTrain(dest) // get trained PC
        transformed := PrincipalComponentTransform(pcModel, dest, 1)
        v := transformed.RawMatrix().Data

        mean, std := stat.MeanStdDev(v, nil)
        keepIndices := []int{}
        for i := 0; i < rlen; i++ {
                if v[i] < mean+selectionStandardDeviation*std && v[i] > mean-
selectionStandardDeviation*std {
                        if rand.Float32() < selectionRate {
                                keepIndices = append(
                        }
                } else {
                        keepIndices = append(keepIndices, i)
                }
        }
        newDest := [][]float64{}
        for ii := range keepIndices {
                pos := keepIndices[ii]
                newDest = append(newDest, d[pos])
        }
        return newDest
}
```

Principle Component transform

Select those data within selectionStandardDeviation from mean with a selectionRate %.
Any data outside selectionStandardDeviation is considered dissimilar data and must be used in the downsampled dataset

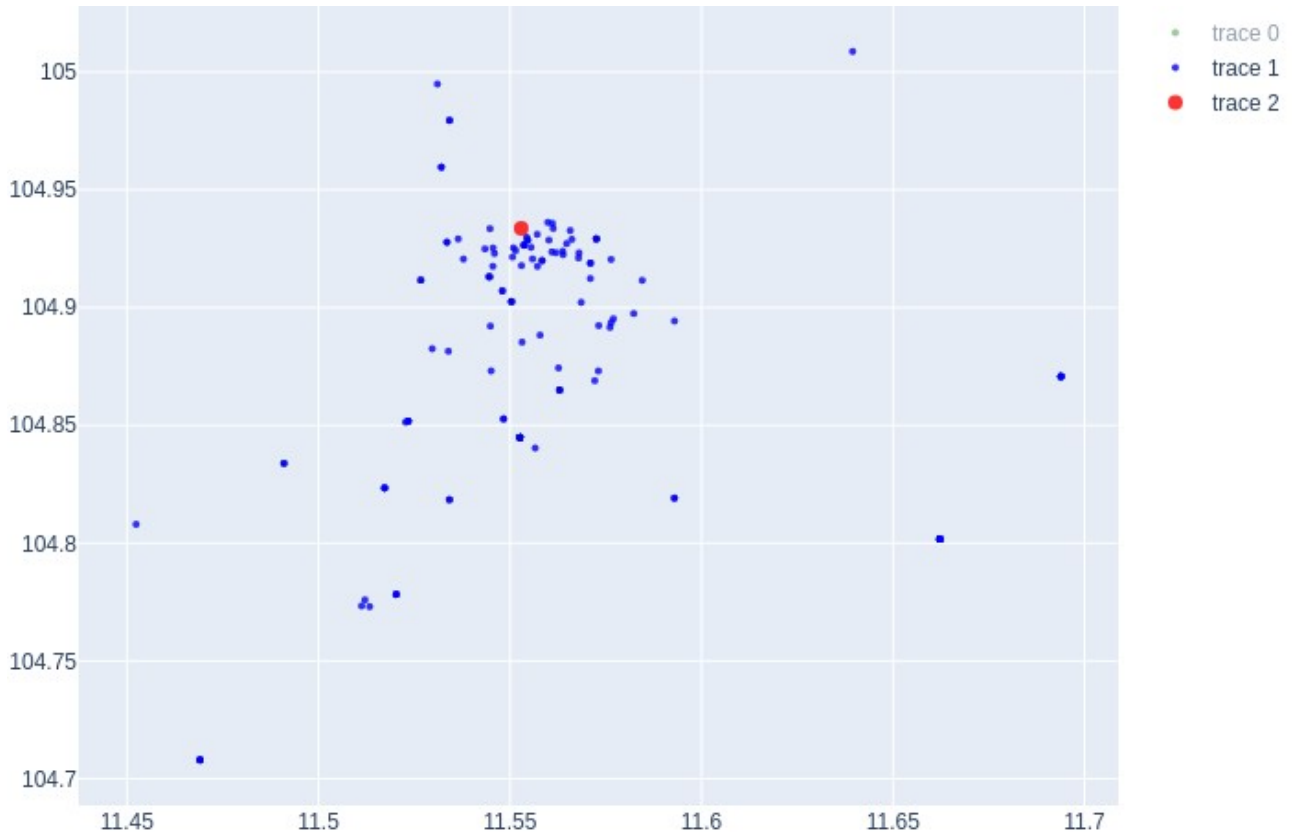


Fig. 3: Visualization of downsampled data(explore2.py)

2.2. Partition data to N worker count in a radial manner with evenly distributed points

The search space for decently shorter path search for N workers at this point is still too big. Hence, it may be ideal if we start the path search on some pre-optimized partitioning heuristics.
The idea is that, the points covered by each worker for decently shorter paths are rather compartmentalized so they have a low chance of cutting through the paths of adjacent workers.
We can work out an estimate of this boundary region for each worker using radial lines for boundary lines for each partition.
We then optimize the angles of these radial lines towards an equal number of nodes for each partiton.

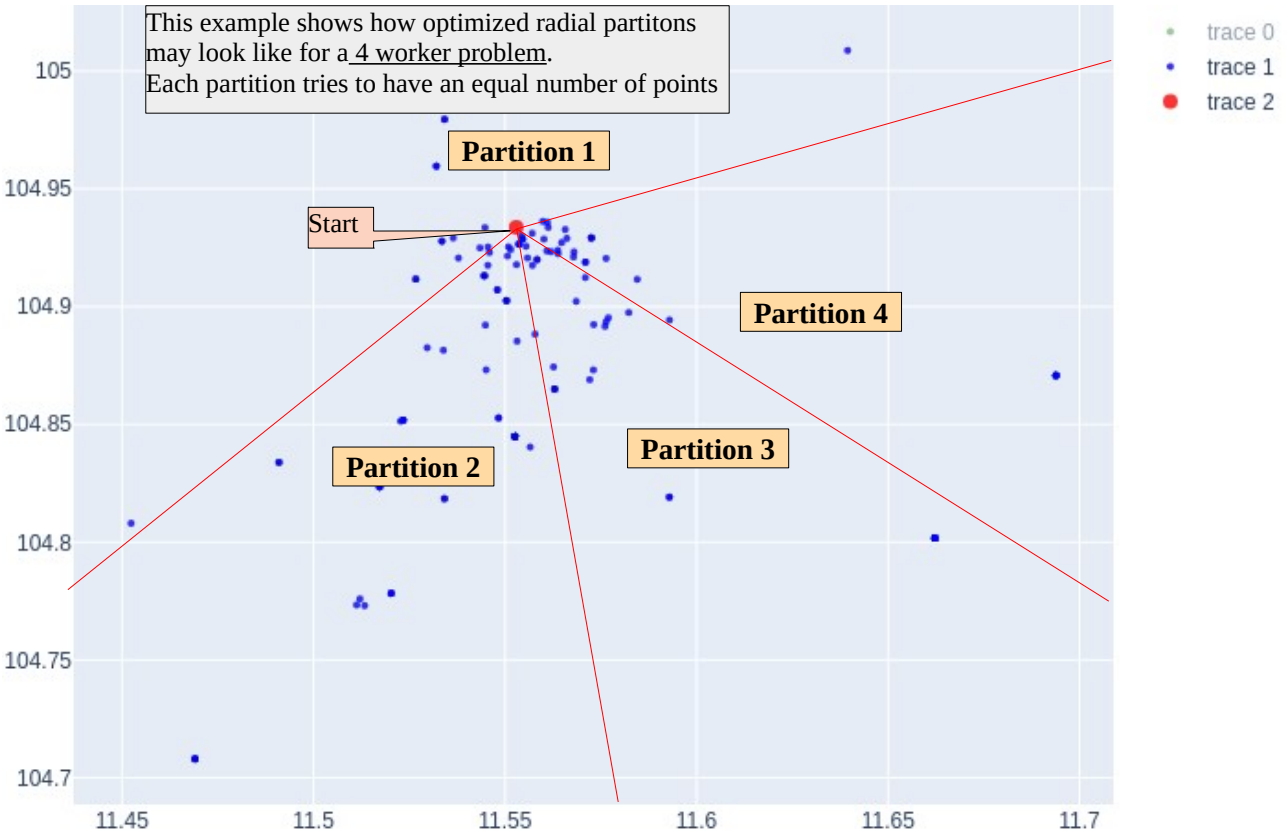
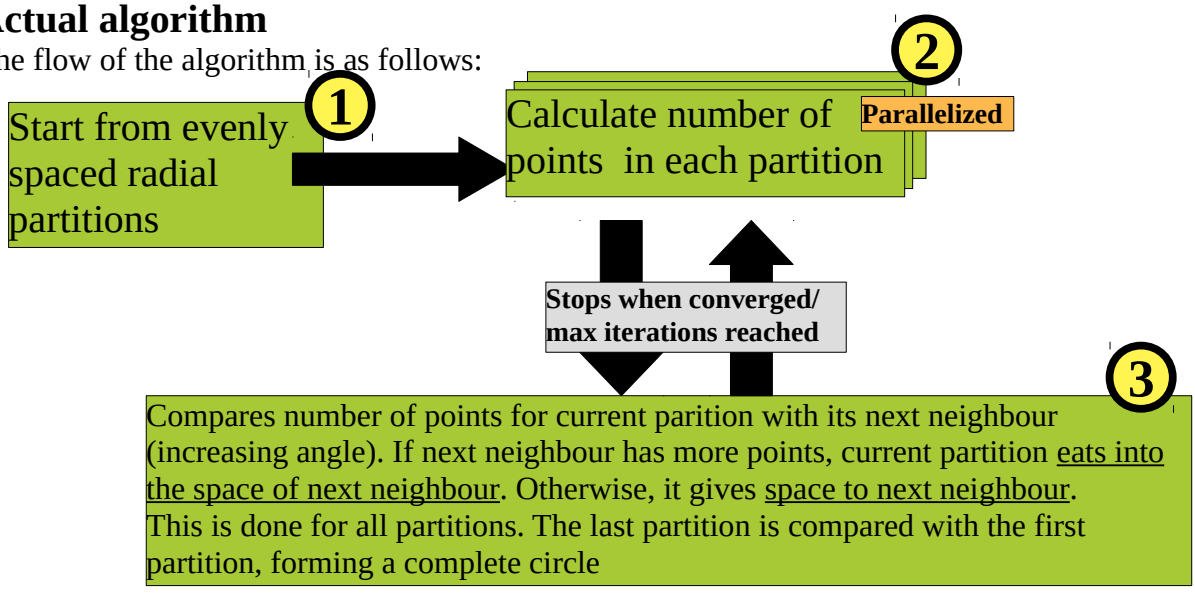


Fig. 4: Optimized radial partitioning for 4 workers

Actual algorithm

The flow of the algorithm is as follows:



Code

pathfinding.go/optimizerAnglePartition

```
func optimizerAnglePartition(points [][]float64, partitionCount int) []AnglePartition {
	// Reduce total distance
	wg := sync.WaitGroup{}
	intervalAngle := fullRad / float64(partitionCount)
	pointsAngle := _getAnglePoints(points)
	partitions := []AnglePartition{}
	for i := 0; i < partitionCount; i++ {
		partitions = append(partitions, AnglePartition{start: float64(i) * intervalAngle, end: (float64(i) + 1) *
intervalAngle})
	}
	for i := range partitions {
		wg.Add(1)
		go _getPartitionPoints(&partitions[i], pointsAngle, &wg)
	}
	wg.Wait()
	k := 1.0
	for iii := 0; iii < 100; iii++ { //converges in ~100
		for i := range partitions {
			nextP := i + 1
			if i+1 == len(partitions) {
				nextP = 0
			}
			ptdiff := len(partitions[nextP].pointsPos) - len(partitions[i].pointsPos)
			avchng := 0.0
			if ptdiff > 0 { //eats into next
				nextAvRng := partitions[nextP].end - partitions[nextP].start
				avchng = float64(ptdiff) / 2.0 / float64(len(partitions[nextP].pointsPos)) *
nextAvRng

			} else if ptdiff < 0 { //eats into cur
				curAvRng := partitions[i].end - partitions[i].start
				avchng = float64(ptdiff) / 2.0 / float64(len(partitions[i].pointsPos)) *
g

			}
			partitions[nextP].start += k * avchng
			if partitions[nextP].start >= fullRad {
				partitions[nextP].start -= fullRad
				partitions[nextP].end -= fullRad
			} else if partitions[nextP].start < 0 {
				partitions[nextP].start += fullRad
				partitions[nextP].end += fullRad
			}
			partitions[i].end += k * avchng
		}
		k *= 0.99
		for i := range partitions {
			print(len(partitions[i].pointsPos))
			print(",")
		}

		println("iter:", iii)
		for i := range partitions {
			wg.Add(1)
			go _getPartitionPoints(&partitions[i], pointsAngle, &wg)
		}
		wg.Wait()
	}
	return partitions
}
```

Create an array of partitions for N workers with each partition equal angle apart

Calculate number of points for each partition using Goroutines (parallelized)

Max iteration of 100

"k" is a decaying multiplier for angle changes

Comparison of points with next neighbour is done here. Angles adjust based on whether current partition has more or less points than its next neighbour

Sample output for 4 worker problem:

```
3,12,198,144,iter: 0
40,85,106,126,iter: 1
67,91,105,94,iter: 2
74,93,103,87,iter: 3
77,104,82,94,iter: 4
93,85,108,71,iter: 5
41,92,85,139,iter: 6
72,85,110,90,iter: 7
80,95,90,92,iter: 8
...
89,89,90,89,iter: 97
89,90,89,89,iter: 98
89,89,90,89,iter: 99
```

For last 3 iterations, partition sizes are unchanged. The results have converged.

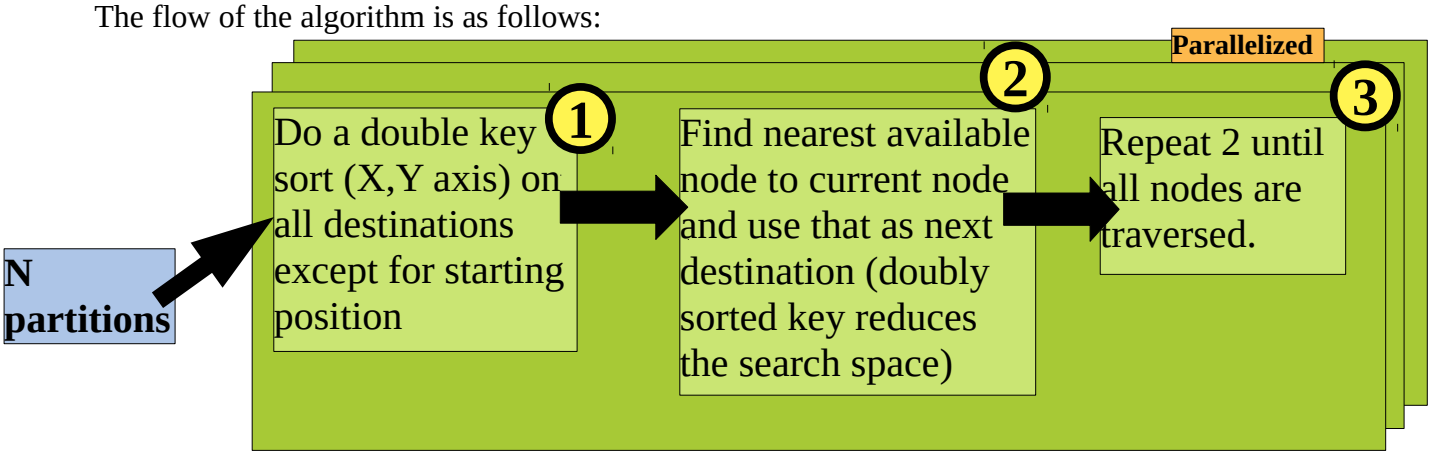
Columns represent different workers/partitions.
Number of points for partition 1 at each iteration.

2.3. Implement actual path search algo for each partition

Using the partitioned points from 2.2, we can implement a path search for each worker/partition.
In this solution, a naive path search using nearest neighbour approach was used.
This means the solution is **not optimal, but reasonable**.
However, it is recommended to look for better existing solutions for this (will need time to find Golang implementations, or will have to write my own)

Actual algorithm

The flow of the algorithm is as follows:



pathfinding.go/_findShortestDistanceWalk

```
func _findShortestDistanceWalk(startpt []float64, points [][]float64) [][]float64 {
        // this assumes walk from 0,0
        // double index, sort 1st index, find nearest neigh
        // branch-and-bound
        sort.Slice(points, func(i, j int) bool {
                return points[i][1] < points[j][1]
        })
        sort.Slice(points, func(i, j int) bool {
                return points[i][0] < points[j][0]
        })

        start := [][]float64{startpt}
        points = append(start, points...)
        path := make([]int, len(points))
        used := make([]bool, len(points))

        used[0] = true
        for ii := 0; ii < len(points)-1; ii++ {
                curPathPos := ii
                curNode := path[curPathPos]
                shortestDist := -1.0

                for i := 1; i < len(points); i++ {
                        if !used[i] {
                                if shortestDist < 0 {
                                        shortestDist = _distL2(points[curNode], points[i])
                                        path[curPathPos+1] = i
                                } else if math.Abs(points[curNode][0]-points[i][0]) < shortestDist {
                                        d := _distL2(points[curNode], points[i])
                                        if d < shortestDist {
                                                path[curPathPos+1] = i
                                                shortestDist = d
                                        }
                                } else if points[i][0]-points[curNode][0] > shortestDist { //first index already > current shortest dist
ignore rest
                                        break
                                }
                        }
                }
                used[path[curPathPos+1]] = true
        }
        pathNodes := [][]float64{}
        for i := range path {
                pathNodes = append(pathNodes, points[path[i]])
        }
        return pathNodes[1:]
}
```

Double sort

Nearest neighbour path search using L2 Norm

Sample output for 4 worker problem:

Zero centered nearest neighbour path of worker 0
-9.881747e-004 -1.522353e-003
-2.181221e-003 -2.666762e-003
-2.916504e-003 -9.577773e-004
+9.859312e-004 +1.720140e-003
+3.196548e-003 +4.231431e-004
...

All these are coordinates relative to the starting point

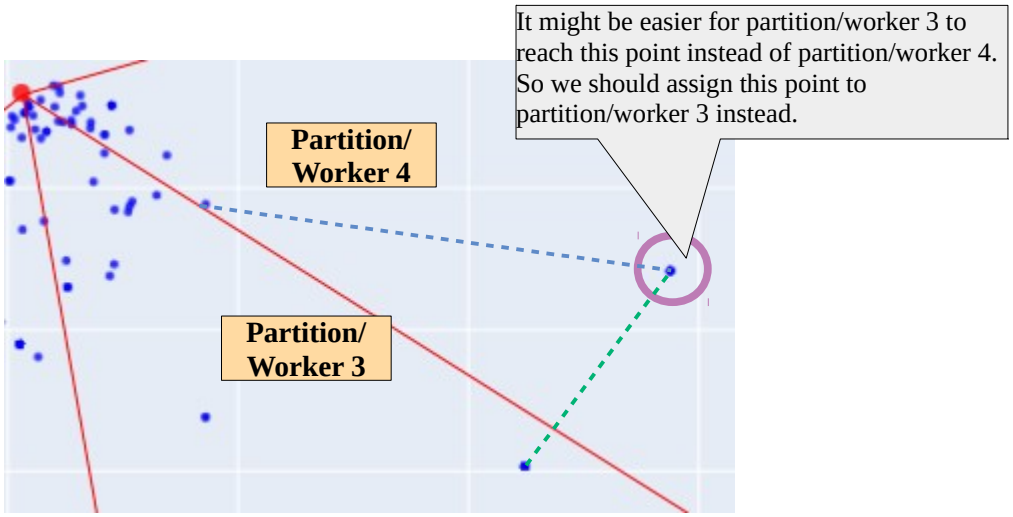
Zero centered nearest neighbour path of worker 1
+5.390953e-003 -2.216627e-003
+5.390953e-003 -2.216627e-003
+7.231544e-003 -4.993727e-003
+6.806206e-003 -5.695631e-003
+6.806206e-003 -5.695631e-003

Zero centered nearest neighbour path of worker 2
+2.107452e-003 -2.857497e-003
+2.107452e-003 -2.857497e-003
+2.108406e-003 -2.857497e-003
+1.387428e-003 -3.788283e-003
+1.387428e-003 -3.788283e-003
...

Zero centered nearest neighbour path of worker 3
-3.444445e-004 -5.131056e-003
+7.169950e-004 -7.145216e-003
+7.169950e-004 -7.145216e-003
-8.313776e-005 -8.564284e-003
-1.050723e-004 -8.564284e-003
...

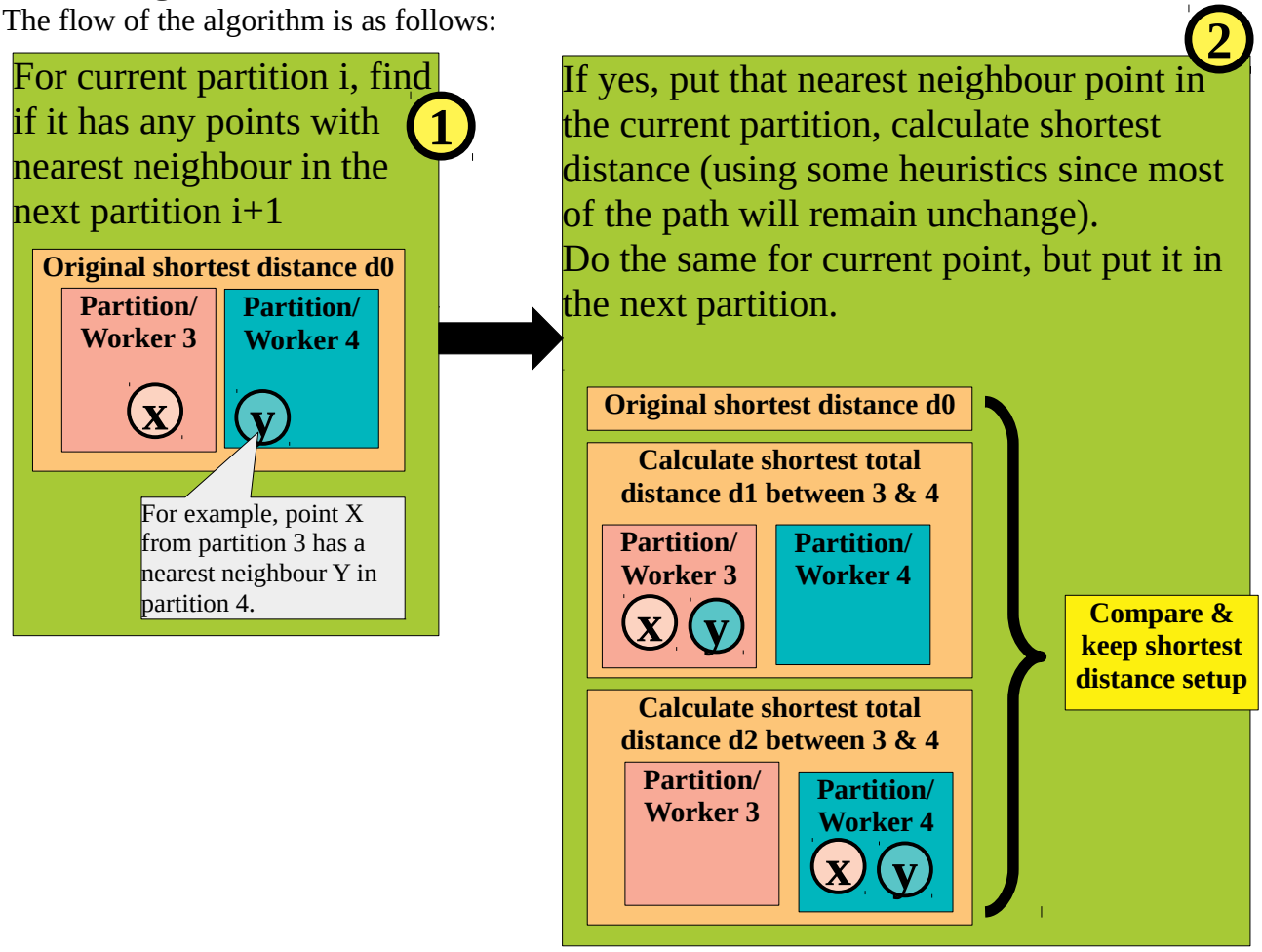
2.4. Optimize the total distance travelled for each worker and their adjacent neighbours

While decently shorter path is implemented within each partition, the total decently shorter travelled distance by all the workers is still unoptimized. Hence we need this step to reduce global decently shorter travelled distance.



Actual algorithm

The flow of the algorithm is as follows:

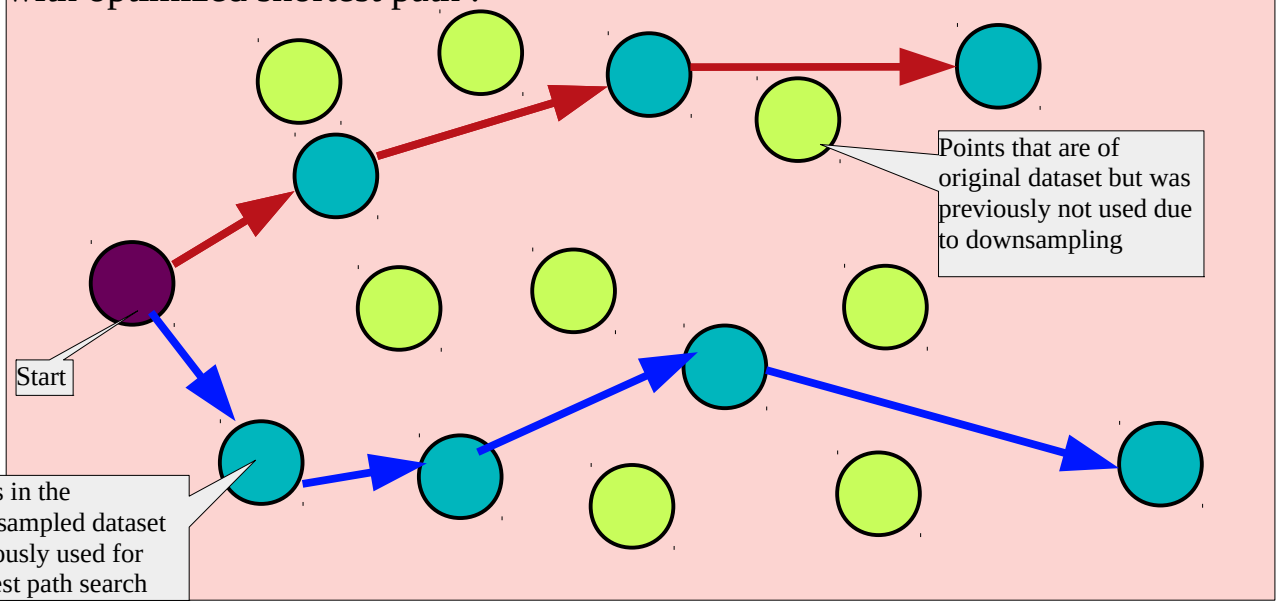


Actual code not yet implemented.

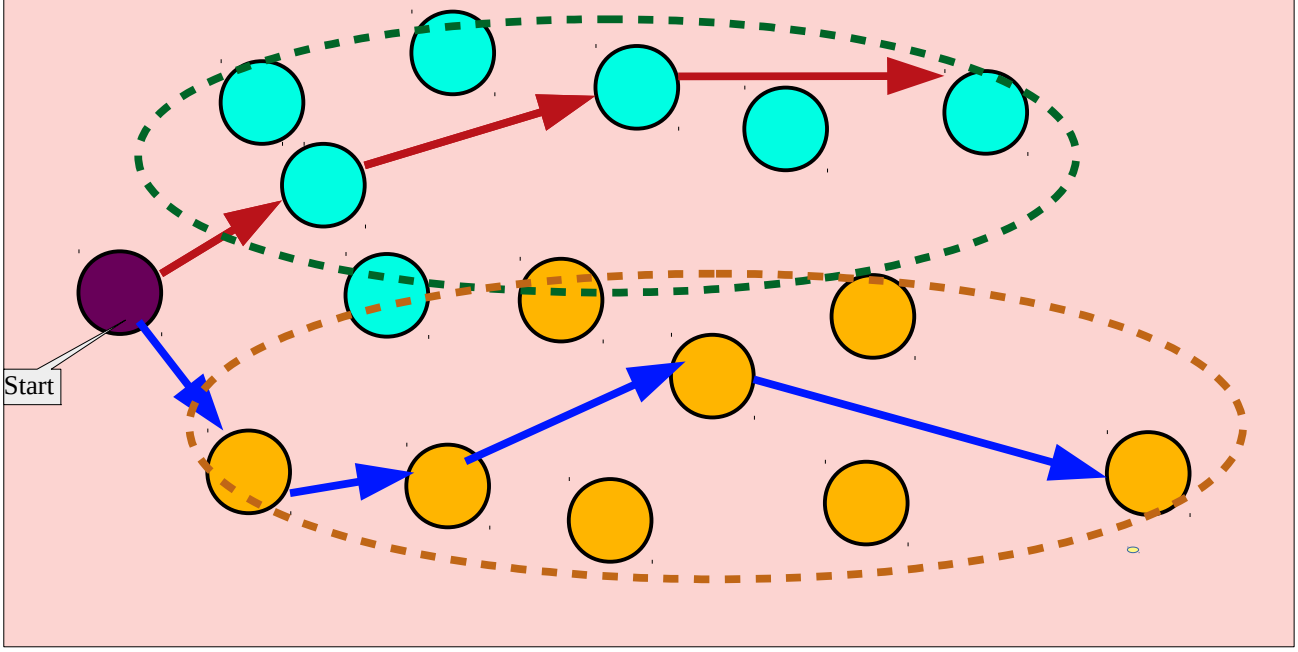
2.5. Use optimized path search from downsampled data to find shorter paths for original data

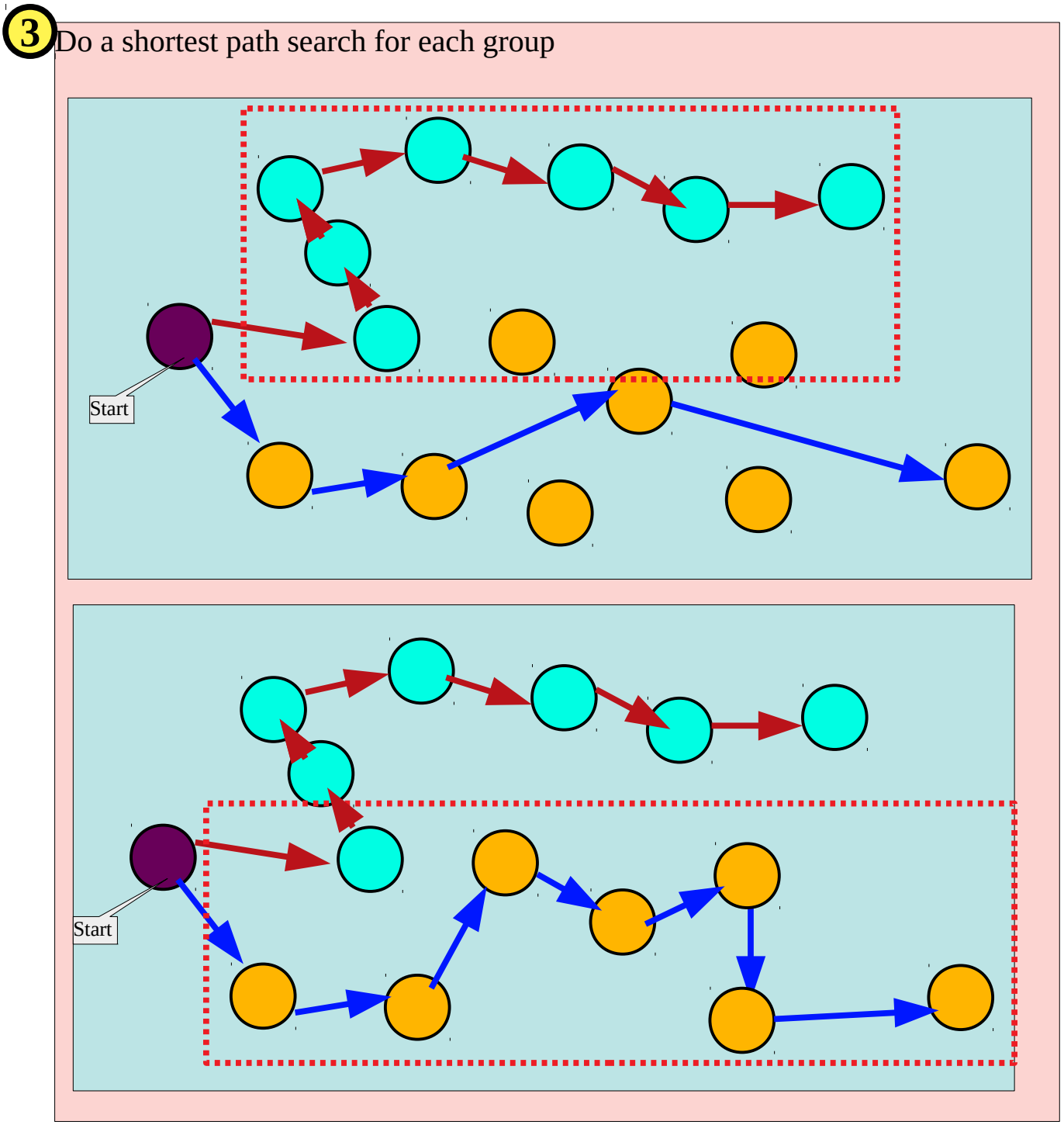
Because we used downsampled points to speed up optimization, the result is not entirely reflective of the actual solution.
Now we have an optimized solution from the downsampled points, we use this result to help us find our shortest paths solution for the actual dataset.

1. Added the unsampled data from original dataset (lime) to downsampled data with optimized shortest path .



2. Create locally nearest neighbour groups where each previously unsampled point is grouped together with its nearest downsampled point non-starting point(A starting point is common to all paths, so it is pointless to put any points together with it) along with all other downsampled points that share the same path





This will yield a decently “Shorter” path search. To ensure better results, we can perform optimization across neighbouring partitions again as we did in 2.5.

Actual code not yet implemented.

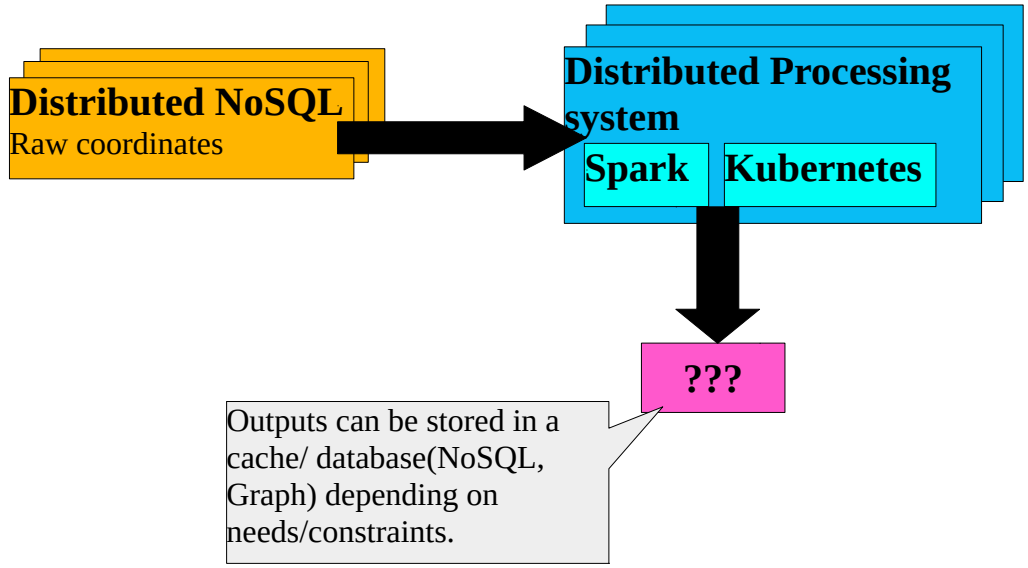
2.6. Summary

The steps taken from 2.1. to 2.5. drastically reduces search space and utilizes parallelism to speed up solution search. The final solution would be a decently shorter total distance travelled. To find actual shortest distance will require computing all possible path permutations for a NP-hard problem, which may not worth to find in reasonable time/resources.

3. Scaling up

1) By using a downsampled approach to this problem, we have added a possibly useful technique to reduce search space for much bigger problems. (eg. Finding decently shorter paths with 1 million data points and 1000 workers but using only 10,000 downsampled points to help us find a decently good solution).

2) Depending on real world constrains/needs, we may be able to change parallelized logic to distributed codes across multiple machines (Spark), as we could also do with the data storage (possibly distributed NoSQL for raw coordinates)



4. Other recommendations

1) Data sanitization should be done upstream. Eg. Coordinates should only contain floats if they are stored in a database.

2) Path search solution does not take into account other real life variables such as traffic conditions, laws, personal travelling preference, renumeration package et cetera. So it should not be treated as the only way for delivering packages.