TSK - Traveling Salesman on KMeans Clusters

Ashar Nadeem and Matthew Goodman

Table of Contents

Background

K-Means clustering is an NP-hard problem that partitions coordinates, or nodes, into clusters with the mean of the cluster serving as its prototype, or centroid. Traveling Salesman is an NP-hard problem where the shortest path between coordinates is found under the constraint that each coordinate is visited at least once.

These problems can be combined together to create an algorithm that creates efficient paths within a list of inputs by clustering them together by proximity and then finding the optimal path within each cluster. More optimized and sophisticated versions of these algorithms are used at marketplace companies like DoorDash and Veho, and this algorithm is explored in a more simple manner in this paper including the effects of parallel implementation.

High Level Approach

It is more efficient to run traveling salesman on a small number of coordinates as opposed to very large numbers that increase the time complexity as more inputs are provided. This especially holds true for the brute force method to solving the TSP, and thus it makes more sense to break the inputs into clusters on which TSP can be run, finding many sub paths that are more optimal than a single traveling salesman traversing the entire list of coordinates.

The number of clusters, *k*, is provided by the user. The coordinates are hardcoded as the latitude and longitude of every capital city in the United States, and K-Means clustering is applied to said group of coordinates, breaking down the large input into smaller, tightly coupled clusters that make for more efficient paths.

The output of this K-Means algorithm is then fed into TSP, which computes the optimal path within the cluster and outputs the list of coordinates, in order, that result in said optimal path.

The total output of the program is a list of paths which are the optimal way to visit each coordinate given the input constraints.

Traveling Salesman

Implementation

tsk - TS.hs

```haskell
module TS
  ( tsp,
  )
where

import Data.List (minimumBy, permutations)
import Data.Ord (comparing)
import Utils (Coordinate, cost)

-- get the shortest path
tsp :: [Coordinate] -> [Coordinate]
tsp coordinates = minimumBy (comparing cost) (permutations coordinates)
```

The approach to traveling salesmen is one of brute force. An input is given to the function which is a list of coordinates that represent the cluster we are trying to optimize for. Every possible permutation of said coordinates is found, and the cost of traversing each permutation in the order it is in is compared. The least costful permutation is then taken as the optimal path amongst the cluster.

Serial

tsk - Lib.hs

```haskell
let shortestPaths = map (tsp . snd) clusters
```

When running the program serially, each cluster is mapped over and TSP is applied to the cluster. This results in threads serially running TSP on 1 cluster at a time and returning the output.

Parallel

tsk - Lib.hs

```haskell
let shortestPaths = parMap rpar (tsp . snd) clusters
```

When running the program in parallel, each cluster is mapped over in parallel, with TSP being run on the clusters simultaneously as each thread can run the program. The final output is returned.

K Means

tsk - KMeans.hs

```haskell
module KMeans
  ( kmeans,
  )
where

import Data.List (minimumBy, sort, transpose)
import Data.Map
  ( Map,
    empty,
    foldrWithKey,
    fromListWith,
    insert,
    keys,
  )
import Data.Ord (comparing)
import Utils (Coordinate, distance, marginOfError)

-- assign coordinate to a cluster
assign :: [Coordinate] -> [Coordinate] -> Map Coordinate [Coordinate]
assign centroids coordinates = fromListWith (++) [(assignCoordinate c, [c]) | c <- coordinates]
  where
    assignCoordinate c = minimumBy (comparing (distance c)) centroids

-- recalculate clusters
recalculate :: Map Coordinate [Coordinate] -> Map Coordinate [Coordinate]
recalculate = foldrWithKey insertCenter empty
  where
    insertCenter _ xs = insert (center xs) xs
    center [] = [0, 0]
    center ps = map average (transpose ps)
    average xs = sum xs / fromIntegral (length xs)

-- run kmeans until centroids no longer move within margin of error
kmeans :: [Coordinate] -> [Coordinate] -> Map Coordinate [Coordinate]
kmeans centroids coordinates =
  if converged
    then clusters
    else kmeans (keys clusters) coordinates
  where
    converged = all (< marginOfError) $ zipWith distance (sort centroids) (keys clusters)
    clusters = recalculate (assign centroids coordinates)
```

A basic K-Means Clustering algorithm is used that takes inputs of coordinates, and calculates centroids for the list that it operates on. As it traverses the list, it assigns coordinates to the closest centroid it finds, while recalculating the cluster that had a coordinate added to it. This continues as all coordinates are added to a cluster, and then the program continues to run to optimize the clusters until the centroids no longer move within a margin of error, signifying that an optimal build of clusters has been achieved.

Utilities

tsk - Utils.hs

```haskell
module Utils
  ( Coordinate,
    cost,
    marginOfError,
    distance,
    stateCapitals,
  )
where

-- data type of a coordinate
type Coordinate = [Double]

marginOfError :: Double
marginOfError = 0.00001

-- euclidean distance between 2 coordinates
distance :: Coordinate -> Coordinate -> Double
distance x y = sqrt $ sum $ map (^ (2 :: Integer)) $ zipWith (-) x y

-- cost/distance of path
cost :: [Coordinate] -> Double
cost coordinates = sum $ zipWith distance coordinates (tail coordinates)
```

Common utility functions include the type that represents a coordinate, the margin of error that is agreed upon, a function to calculate the euclidean distance between two coordinates, as well as the cost to traverse a path of coordinates.

Main

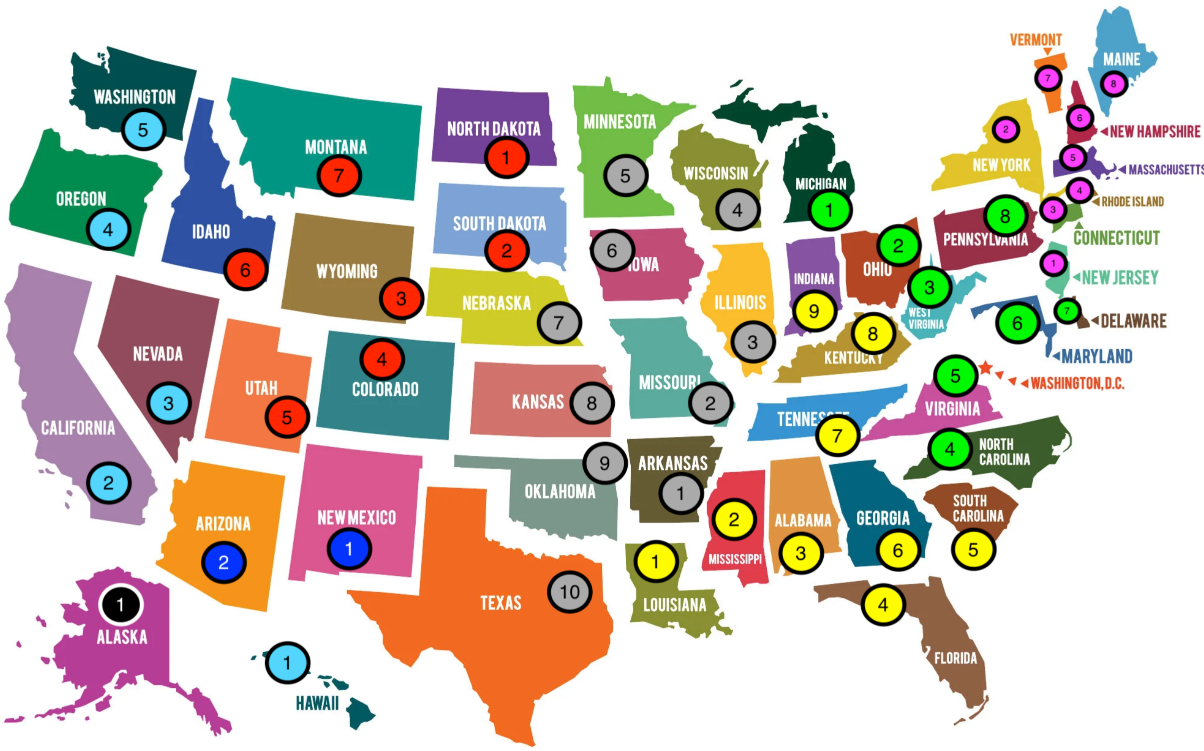
tsk - Lib.hs

```haskell
module Lib (runTsk) where

import Control.Monad (unless, when)
import Control.Parallel.Strategies (parMap, rpar)
import Data.Map (toList)
import KMeans (kmeans)
import System.Environment (getArgs, getProgName)
import System.Exit (die)
import TS (tsp)
import Utils (stateCapitals)

runTsk :: IO ()
runTsk = do
  program <- getProgName
  programArgs <- getArgs
  when (length programArgs /= 2) $
    die ("usage: " ++ program ++ " <strategy> <num_clusters>")
  let (strategy : rawK : _) = programArgs
  let k = read rawK
  unless (strategy == "s" || strategy == "p") $
    die ("usage: " ++ program ++ " <strategy> <num_clusters>")
  let clusters = toList $ kmeans (take k stateCapitals) stateCapitals
  if strategy == "s"
    then do
      let shortestPaths = map (tsp . snd) clusters
      mapM_ print shortestPaths
    else do
      let shortestPaths = parMap rpar (tsp . snd) clusters
      mapM_ print shortestPaths
```

Input

The standard input used to run all performance tests consisted of 3 things. Program arguments were passed in to determine whether the code shall be run serially or in parallel, and the number of clusters to create was passed in as well. Lastly, the standard dataset on which to run the code was hardcoded, consisting of the latitude and longitude of every capital city in the United States. The dataset was sourced from here.

Output



For visualization purposes, the list of optimal routes output by the algorithm was overlaid on top of a map of the United States. The denseness is immediately obvious through the color coded clusters, and the optimal routes amongst the individual clusters passes the eye test when looking through numbered nodes and where the obvious next stop should be. As mentioned earlier, it is not imperative that a route start and end at the same point. Overall, the output shows satisfactory results and there is confidence in the effectiveness of the algorithm to solve the problem at hand.

Performance

Serial

tsk - output.txt

````
run command:
```
time stack exec tsk-exe s 8
```

output:
```
[[30.457069,-91.187393],[32.303848,-90.182106],[32.377716,-86.300568],
[30.438118,-84.281296],[34.000343,-81.033211],[33.749027,-84.388229],
[36.16581,-86.784241],[38.186722,-84.875374],[39.768623,-86.162643]]
[[35.68224,-105.939728],[33.448143,-112.096962]]

[[21.307442,-157.857376],[38.576668,-121.493629],[39.163914,-119.766121],
[44.938461,-123.030403],[47.035805,-122.905014]]

[[34.746613,-92.288986],[38.579201,-92.172935],[39.798363,-89.654961],
[43.074684,-89.384445],[44.955097,-93.102211],[41.591087,-93.603729],
[40.808075,-96.699654],[39.048191,-95.677956],[35.492207,-97.503342],
[30.27467,-97.740349]]

[[42.733635,-84.555328],[39.961346,-82.999069],[38.336246,-81.612328],
[35.78043,-78.639099],[37.538857,-77.43364],[38.978764,-76.490936],
[39.157307,-75.519722],[40.264378,-76.883598]]

[[40.220596,-74.769913],[42.652843,-73.757874],[41.764046,-72.682198],
[41.830914,-71.414963],[42.358162,-71.063698],[43.206898,-71.537994],
[44.262436,-72.580536],[44.307167,-69.781693]]

[[46.82085,-100.783318],[44.367031,-100.346405],[41.140259,-104.820236],
[39.739227,-104.984856],[40.777477,-111.888237],[43.617775,-116.199722],
[46.585709,-112.018417]]

[[58.301598,-134.420212]]
```

timing:
```
stack exec tsk-exe s 8  6.39s user 1.08s system 242% cpu 6.837 total
```
````

Parallel

tsk - output.txt

````
run command:
```
time stack exec tsk-exe p 8
```

output:
```
[[30.457069,-91.187393],[32.303848,-90.182106],[32.377716,-86.300568],
[30.438118,-84.281296],[34.000343,-81.033211],[33.749027,-84.388229],
[36.16581,-86.784241],[38.186722,-84.875374],[39.768623,-86.162643]]
[[35.68224,-105.939728],[33.448143,-112.096962]]

[[21.307442,-157.857376],[38.576668,-121.493629],[39.163914,-119.766121],
[44.938461,-123.030403],[47.035805,-122.905014]]

[[34.746613,-92.288986],[38.579201,-92.172935],[39.798363,-89.654961],
[43.074684,-89.384445],[44.955097,-93.102211],[41.591087,-93.603729],
[40.808075,-96.699654],[39.048191,-95.677956],[35.492207,-97.503342],
[30.27467,-97.740349]]

[[42.733635,-84.555328],[39.961346,-82.999069],[38.336246,-81.612328],
[35.78043,-78.639099],[37.538857,-77.43364],[38.978764,-76.490936],
[39.157307,-75.519722],[40.264378,-76.883598]]

[[40.220596,-74.769913],[42.652843,-73.757874],[41.764046,-72.682198],
[41.830914,-71.414963],[42.358162,-71.063698],[43.206898,-71.537994],
[44.262436,-72.580536],[44.307167,-69.781693]]

[[46.82085,-100.783318],[44.367031,-100.346405],[41.140259,-104.820236],
[39.739227,-104.984856],[40.777477,-111.888237],[43.617775,-116.199722],
[46.585709,-112.018417]]

[[58.301598,-134.420212]]
```

timing:
```
stack exec tsk-exe p 8  4.27s user 0.65s system 110% cpu 4.458 total
```
````

Conclusion

Both implementations yield identical results that pass the eye test in terms of efficiency and correctness.The serial implementation takes ~6.39 seconds whereas the parallel implementation takes ~4.27 seconds. The parallel implementation improves performance by 33.2%, a substantial improvement that could be further optimized with more time and effort in finding better solutions to TSP, primarily anything that is not brute force.