Student Names: Joe Parker, Trikay Nalamada
Student IDs: jep2214, tn2466

Fast, Functional Traveling Salesman

Final Project

---

1. The Traveling Salesman Problem

   The TSP is a classic optimization problem that involves finding the shortest possible route that visits a given set of cities and returns to the starting city. It is called the "Traveling Salesman Problem" because it is often used to model the problem of a salesman who must visit a set of cities and return home, minimizing the total distance traveled.

   In the TSP, a set of cities is given, along with the distances between each pair of cities. The goal is to find a route that visits each city exactly once, starting and ending at a specific city (called the "home" or "depot" city), and minimizing the total distance traveled. The TSP can be formulated as a graph, where the cities are represented by vertices and the distances between cities are represented by edges.

   The TSP is an NP-hard problem, meaning that it is difficult to find an exact solution for large instances of the problem in a reasonable amount of time. This makes it challenging to solve the TSP optimally, especially for instances with a large number of cities. As a result, many algorithms have been developed to find approximate solutions to the TSP, such as heuristics, metaheuristics, and approximation algorithms. These algorithms are designed to find good solutions quickly, rather than optimal solutions.

   There are many variations of the TSP, including the symmetric TSP (where the distance between two cities is the same in both directions) and the asymmetric TSP (where the distance between two cities may be different in each direction). The TSP has numerous real-world applications, such as routing delivery trucks, scheduling airline routes, and planning sales territories. It is also a popular problem in computer science and operations research for testing and comparing the performance of different optimization algorithms.

2. The Brute Force Approach

   The brute force approach to solving the Traveling Salesman Problem (TSP) involves enumerating all possible routes and selecting the shortest one.

   In the brute force approach, the algorithm generates all possible permutations of the cities, calculates the total distance traveled for each permutation, and selects the permutation with the shortest distance. This is a simple and straightforward approach, but it becomes impractical for large instances of the TSP due to the exponential number of possible permutations.

   For example, consider a TSP with 5 cities. There are $5! = 120$ possible permutations of the cities, which can be generated by rearranging the cities in different orders. If we add one more city, the number of permutations increases to $6! = 720$. As the number of cities increases, the number of permutations grows exponentially, making the brute force approach impractical for large instances of the TSP.

   Despite its simplicity and straightforwardness, the brute force approach is not usually used to solve the TSP in practice due to its high time complexity. Instead, more efficient algorithms, such as heuristics, metaheuristics, and approximation algorithms, are used to find good solutions to the TSP in a reasonable amount of time.

```
import Data.List
import Control.Parallel.Strategies

graphMaker :: Int -> [(Float, Float)]
graphMaker num = [(fromIntegral i, fromIntegral i)| i <- [0..num]]

graph :: [(Float, Float)]
graph = graphMaker 9

distance :: Floating a => (a, a) -> (a, a) -> a
distance (x1 , y1) (x2 , y2) = sqrt (x'*x' + y'*y')
    where
      x' = x1 - x2
      y' = y1 - y2

map_traverse :: (Floating b) => [(b,b)] -> b
map_traverse [p1, p2] = distance p1 p2
map_traverse (p1:p2:ps) = distance p1 p2 + map_traverse (p2 : ps)
map_traverse [_] = 0.0
map_traverse [] = 0.0


allPerms :: [[(Float, Float)]]
allPerms = permutations (tail graph)


traverseAll :: [(Float, [(Float, Float)])]
traverseAll = map (\x -> (map_traverse (addEnds x (head graph)), x)) allPerms


addEnds :: [a] -> a -> [a]
addEnds xs end = end:xs ++ [end]


shortest :: (Float, [(Float, Float)])
shortest = minimum traverseAll


main :: IO ()
main = do
        print shortest

}
```

The main sequential parts of the code that are causing huge delays are the generations of the allPerms list (which is the list of all permutations of the graph) and the traversal of these permutations in a sequential manner. Specifically allPerms and traverseAll need to be optimized.

3. Parallelized Brute Force Approach

   The following pieces of code need to be optimized.

```
allPerms :: [[(Float, Float)]]
allPerms = permutations (tail graph)
```

```
traverseAll :: [(Float, [(Float, Float)])]
traverseAll = map (\x -> (map_traverse (addEnds x (head graph)), x)) allPerms

}
```

We attempt to parallelize them by using different combinations of parList and parListChunk with rpar and rseq and carry out an analysis in the Results section. An example of the code block when using parListChunk on the traverseAll list looks like:

```
allPerms :: [[(Float, Float)]]
allPerms = permutations (tail graph)

traverseAll :: [(Float, [(Float, Float)])]
traverseAll = map (\x -> (map_traverse (addEnds x (head graph)), x)) allPerms `using`
    parListChunk 8 rseq

}
```

4. The HeldKarp Algorithm

   In the TSP, we are given a set of cities 1, 2, 3, ..., n, and we are trying to find the shortest possible route that visits each city exactly once and returns to the starting city. We can represent this route as a permutation of the cities: (1, 2, 3, ..., n). The distance between any two cities i and j is given by a distance function d(i, j).

   The Held-Karp algorithm works by dividing the TSP into smaller subproblems and then using the solutions to these subproblems to find the solution to the overall problem. Specifically, the algorithm uses dynamic programming to solve the TSP by considering all possible combinations of cities that could be visited in a given order.

   To understand how the Held-Karp algorithm works, it is helpful to define some additional notation. Let S be a subset of cities, and let j be a city in S. We can define a function C(S, j) as the shortest possible route that visits all the cities in S, ending at city j.

   With this notation, we can now describe the Held-Karp algorithm as follows:

   (a) Initialize the base case: C(1, 1) = 0, which means the shortest possible route that visits only city 1 and ends at city 1 has a distance of 0.

   (b) For each subset S of cities of size k, where $2 \leq k \leq n$:

   (c) For each city j in S: C(S, j) = min C(S - j, i) + d(i, j), where i is a city in S - j

   (d) To find the shortest possible route that visits all the cities and returns to the starting city, we can use the function C(S, j) where S is the set of all cities and j is the starting city. The shortest route is given by:

$$C(S, 1) = \min C(S - 1, j) + d(j, 1), \text{where j is a city in } S - 1$$

   This algorithm has a time complexity of $O(n^2 * 2^n)$, which means it can solve TSP instances with a large number of cities in a reasonable amount of time. However, it can be slower than some other algorithms for solving the TSP, such as the Christofides algorithm.

   Our implementation of HeldKarp is as below:

```haskell
import qualified Data.List as List
import Data.Function.FastMemo
import Data.Word
import Data.FloatCast

type Point = (Word32, Word32)

makePoint :: Float -> Float -> Point
makePoint x y = (floatToWord x, floatToWord y)

makeTestSet :: Int -> [Point]
makeTestSet n = [ makePoint (fromIntegral a) ((fromIntegral a) + 1) | a <- [1..n]]

convertFloat :: [Point] -> [(Float, Float)]
convertFloat pts = [(wordToFloat a, wordToFloat b) | (a, b) <- pts]

getDistance :: Point -> Point -> Word32
getDistance (x1, y1) (x2, y2) = floatToWord ((((wordToFloat x1 - wordToFloat x2) ** (2 ::
    Float)) + ((wordToFloat y1 - wordToFloat y2) ** (2 :: Float))) ** (0.5 :: Float))

getPathDistance :: [Point] -> Word32
getPathDistance (p1:p2:[]) = getDistance p1 p2
getPathDistance (p1:p2:xs) = floatToWord (wordToFloat (getDistance p1 p2) + wordToFloat
    (getPathDistance (p2:xs)))
getPathDistance _ = floatToWord 0.0

triples :: Point -> [Point] -> [(Point, Point, [Point])]
triples start_pt points = [(start, end, trav) | start <- points, end <- points, start ==
    start_pt, start /= end, let trav = List.delete start $ List.delete end points]

permShort :: [Point] -> [[Point]]
permShort xs = [ x : [y | y <- xs, y /= x] | x <- xs ]


minTrip :: (Point, Point, [Point]) -> Word32
minTrip (start_pt, end_pt, []) = getDistance start_pt end_pt
minTrip (start_pt, end_pt, (pt1:[])) = getDistance start_pt pt1 + getDistance pt1 end_pt
minTrip (start_pt, end_pt, (pt1:xs)) = floatToWord ( wordToFloat (getDistance pt1 end_pt)
    + wordToFloat (minimum (map (minTrip) [(start_pt, pt1, ps) | ps <- pms])))
                                            where
                                                pms = permShort xs
```

The main crux of the solution is in the minTrip function which divides the problem into subproblems and recursively solves it. The optimal substructure of the motivates a dynamic programming style solution in the next section. Notice that the third argument of minTrip contains the paths of the graph as a list, which is not optimal for hashing due to its ordered structure. Hence we chose to reimplement the solution using sets in the following section.

5. A Set Based HeldKarp Algorithm A set based approach is given as below:

```haskell
import qualified Data.List as List
import Data.Function.FastMemo
import Data.Word
```

```
import Data.FloatCast
import qualified Data.Set as Set


type Point = (Word32, Word32)

makeTestTriple :: Float -> Float -> Float -> Float -> [(Float, Float)] -> (Point, Point,
    Set.Set Point)
makeTestTriple x1 y1 x2 y2 trav = (makePoint x1 y1, makePoint x2 y2, Set.fromList (map
    (\(a,b) -> makePoint a b) trav))

makePoint :: Float -> Float -> Point
makePoint x y = (floatToWord x, floatToWord y)

makeTestSet :: Int -> Set.Set Point
makeTestSet n = Set.fromList [ makePoint (fromIntegral a) ((fromIntegral a) + 1) | a <-
    [1..n]]

convertFloat :: [Point] -> [(Float, Float)]
convertFloat pts = [(wordToFloat a, wordToFloat b) | (a, b) <- pts]

getDistance :: Point -> Point -> Word32
getDistance (x1, y1) (x2, y2) = floatToWord ((((wordToFloat x1 - wordToFloat x2) ** (2 ::
    Float)) + ((wordToFloat y1 - wordToFloat y2) ** (2 :: Float))) ** (0.5 :: Float))

getPathDistance :: Set.Set Point -> Word32
getPathDistance pts | (Set.size pts) < 2 = floatToWord 0.0
                    | otherwise = floatToWord (sum (map wordToFloat (zipWith
    (getDistance) pt_lst pt_tail)))
                                    where
                                        pt_lst = Set.toList pts
                                        pt_tail = tail pt_lst


triples :: Point -> [Point] -> [(Point, Point, [Point])]
triples start_pt points = [(start, end, trav) | start <- points, end <- points, start ==
    start_pt, start /= end, let trav = List.delete start $ List.delete end points]

permShort :: [Point] -> [[Point]]
permShort xs = [ x : [y | y <- xs, y /= x] | x <- xs ]


minTripFib :: (Point, Point, Set.Set Point) -> Word32
minTripFib = memoize $ \(start_pt, end_pt, pts) -> case Set.size pts of
        0 -> getDistance start_pt end_pt
        1 -> floatToWord (wordToFloat (getDistance start_pt first_elem) + wordToFloat
    (getDistance first_elem end_pt))
            where
                first_elem = Set.elemAt 0 pts
        _ -> floatToWord (wordToFloat (getDistance end_pt first_elem) + wordToFloat
    (minimum (map minTripFib [(start_pt, first_elem, Set.fromList ps) | ps <- pms])))
            where
                first_elem = Set.elemAt 0 pts
                pms = permShort . Set.toList . snd $ (Set.splitAt 1 pts)
```

The minTripFib function replicates the logic as the one in the previous section and incorporates

memoization to approach the problem in a top-down manner. The set based DP style algorithm gives great speedups as compared to the above mentioned algorithms. The results are demonstrated in the final section.

6. Parallel Set Based HeldKarp Algorithm

   The HeldKarp code has 2 pieces of sequential processing that take up most of the processing time. First being the permShort list comprehension:

```
permShort :: [Point] -> [[Point]]
permShort xs = [ x : [y | y <- xs, y /= x] | x <- xs ]
```

   And next being the sequential map application of minTripFib to all permutations as shown here:

```
        _ -> floatToWord (wordToFloat (getDistance end_pt first_elem) + wordToFloat
    (minimum (map minTripFib [(start_pt, first_elem, Set.fromList ps) | ps <- pms])))
```

   We parallelize both by using parList and parMap with the rpar strategy.

7. Results

   We run a few tests on a 10 point, 11 point and 12 point graph. We record the total time, total sparks generated, total sparks consumed and total sparks overflowed. The algorithms we use for comparison are the Naive one, Naive one with parList and rpar for parallelization, Naive one with parListChunk with parallelization, HeldKarp list version, HeldKarp set version and HeldKarp set version with parallization.

   10 Point Graph:

Algorithm	Time	Total Sparks	Sparks Converted	Sparks Overflowed
Naive Sequential	63.46 s	-	-	-
Naive Parallel (parList+rpar)	4.6 s	7257600	1699409	2779098
Naive Parallel (parListChunk)	4.28 s	453600	453600	0
HeldKarp (List)	11.06 s	-	-	-
HeldKarp (Set)	0.33 s	-	-	-
HeldKarp (Set + parallel)	0.13 s	224	90	0

   11 Point Graph: (We stop computation at 300 sec and label it as "Time limit exceeded")

Algorithm	Time	Total Sparks	Sparks Converted	Sparks Overflowed
Naive Sequential	> 300 s	-	-	-
Naive Parallel (parList+rpar)	97.82 s	79833600	21508132	29162944
Naive Parallel (parListChunk)	101.4 s	4989600	4989600	0
HeldKarp (List)	34.37 s	-	-	-
HeldKarp (Set)	0.42 s	-	-	-
HeldKarp (Set + parallel)	0.2 s	376	102	0

   12 Point Graph:

Algorithm	Time	Total Sparks	Sparks Converted	Sparks Overflowed
Naive Sequential	> 300 s	-	-	-
Naive Parallel (parList+rpar)	> 300 s	-	-	-
Naive Parallel (parListChunk)	> 300 s	-	-	-
HeldKarp (List)	> 300 s	-	-	-
HeldKarp (Set)	1.85 s	-	-	-
HeldKarp (Set + parallel)	0.58 s	588	99	0

Some key observations from the above experiments:

(a) As the points increase, the time taken for most algorithms increases exponentially/factorially

(b) For the 12 point graph, only HeldKarp (Set) and HeldKarp (Set + parallel) finish computation

(c) Variants of HeldKarp are significantly faster than the variants of the Naive implementation

(d) The Set version of HeldKarp, which incorporates memoization, cuts down a lost of time as compared to the List version

(e) The (parList + rpar) of Naive has a large number of overflowed sparks. This could be due to the work needed, per element of the list being parallelized, being minimal.

(f) Using parListChunk and fixing the number of chunks to 8, helps with controlling the overflowed sparks and offers a most efficient way of parallelizing

8. Code Files

NaiveSeq.hs :

```
-- NB --> In the making of this Naive implementation, we referenced
    https://github.com/travisbrady/haskell-tsp
    module Naive_Seq
    (
      shortest,
      traverseAll
    )
    where
    import qualified Data.List as List

    -- Can change graph easily to a different set of input nodes

    graphMaker :: Int -> [(Float, Float)]
    graphMaker num = [(fromIntegral i, fromIntegral i)| i <- [0..num]]

    graph :: [(Float, Float)]
    graph = graphMaker 10

    -- distance :: (Integral a, Floating b) => (a,a) -> (a,a) -> b
    -- distance p1 p2 = sqrt ((fst (p2) - fst(p1))^^2 + ((snd p2) - (snd p1))^^2)

    distance :: Floating a => (a, a) -> (a, a) -> a
    distance (x1 , y1) (x2 , y2) = sqrt (x'*x' + y'*y')
        where
          x' = x1 - x2
          y' = y1 - y2


    map_traverse :: (Floating b) => [(b,b)] -> b
```

```
map_traverse [p1, p2] = distance p1 p2
map_traverse (p1:p2:ps) = distance p1 p2 + map_traverse (p2 : ps)
map_traverse [_] = 0.0
map_traverse [] = 0.0

allPerms :: [[(Float, Float)]]
allPerms = List.permutations (tail graph)

-- Sequential Traversal
traverseAll :: [(Float, [(Float, Float)])]
traverseAll = map (\x -> (map_traverse (addEnds x (head graph)), x)) allPerms

addEnds :: [a] -> a -> [a]
addEnds xs end = end:xs ++ [end]

shortest :: (Float, [(Float, Float)])
shortest = List.minimum traverseAll
```

NaivePar.hs:

```
module Naive_Par
(
  shortest,
  traverseAll
)
where
import qualified Data.List as List
import Control.Parallel.Strategies

-- Can change graph easily to a different set of input nodes

graphMaker :: Int -> [(Float, Float)]
graphMaker num = [(fromIntegral i, fromIntegral i)| i <- [0..num]]

graph :: [(Float, Float)]
graph = graphMaker 10

-- distance :: (Integral a, Floating b) => (a,a) -> (a,a) -> b
-- distance p1 p2 = sqrt ((fst (p2) - fst(p1))^^2 + ((snd p2) - (snd p1))^^2)

distance :: Floating a => (a, a) -> (a, a) -> a
distance (x1 , y1) (x2 , y2) = sqrt (x'*x' + y'*y')
    where
      x' = x1 - x2
      y' = y1 - y2


-- Can probably implement this with a fold, not sure how though
map_traverse :: (Floating b) => [(b,b)] -> b
map_traverse [p1, p2] = distance p1 p2
map_traverse (p1:p2:ps) = distance p1 p2 + map_traverse (p2 : ps)
map_traverse [_] = 0.0
map_traverse [] = 0.0

allPerms :: [[(Float, Float)]]
allPerms = List.permutations (tail graph) --`using` parList rseq
```

```
    -- Sequential -> rpar (overflowed sparks) -> parListChunk
    traverseAll :: [(Float, [(Float, Float)])]
    traverseAll = map (\x -> (map_traverse (addEnds x (head graph)), x)) allPerms `using`
    parListChunk 8 rseq

    addEnds :: [a] -> a -> [a]
    addEnds xs end = end:xs ++ [end]

    shortest :: (Float, [(Float, Float)])
    shortest = List.minimum traverseAll
```

HeldKarpList.hs:

```
module HeldKarp_List
(makePoint,
makeTestSet,
convertFloat,
getDistance,
getPathDistance,
triples,
permShort,
minTrip,
minTripFib) where

import qualified Data.List as List
import Data.Function.FastMemo
import Data.Word
import Data.FloatCast

type Point = (Word32, Word32)

makePoint :: Float -> Float -> Point
makePoint x y = (floatToWord x, floatToWord y)

makeTestSet :: Int -> [Point]
makeTestSet n = [ makePoint (fromIntegral a) ((fromIntegral a) + 1) | a <- [1..n]]

convertFloat :: [Point] -> [(Float, Float)]
convertFloat pts = [(wordToFloat a, wordToFloat b) | (a, b) <- pts]

getDistance :: Point -> Point -> Word32
getDistance (x1, y1) (x2, y2) = floatToWord ((((wordToFloat x1 - wordToFloat x2) ** (2 ::
    Float)) + ((wordToFloat y1 - wordToFloat y2) ** (2 :: Float))) ** (0.5 :: Float))

getPathDistance :: [Point] -> Word32
getPathDistance (p1:p2:[]) = getDistance p1 p2
getPathDistance (p1:p2:xs) = floatToWord (wordToFloat (getDistance p1 p2) + wordToFloat
    (getPathDistance (p2:xs)))
getPathDistance _ = floatToWord 0.0

triples :: Point -> [Point] -> [(Point, Point, [Point])]
triples start_pt points = [(start, end, trav) | start <- points, end <- points, start ==
    start_pt, start /= end, let trav = List.delete start $ List.delete end points]

permShort :: [Point] -> [[Point]]
```

```
permShort xs = [ x : [y | y <- xs, y /= x] | x <- xs ]


minTrip :: (Point, Point, [Point]) -> Word32
minTrip (start_pt, end_pt, []) = getDistance start_pt end_pt
minTrip (start_pt, end_pt, (pt1:[])) = floatToWord ( wordToFloat (getDistance start_pt
    pt1) + wordToFloat (getDistance pt1 end_pt))
minTrip (start_pt, end_pt, (pt1:xs)) = floatToWord ( wordToFloat (getDistance pt1 end_pt)
    + wordToFloat (minimum (map (minTrip) [(start_pt, pt1, ps) | ps <- pms])))
                                                    where
                                                        pms = permShort xs
```

HeldKarpSet.hs: (minTripFib Contains the memoization)

```
module HeldKarpSet
(makeTestTriple,
makePoint,
makeTestSet,
convertFloat,
getDistance,
getPathDistance,
triples,
permShort,
minTrip,
minTripFib)
where
import qualified Data.List as List
import Data.Function.FastMemo
import Data.Word
import Data.FloatCast
import qualified Data.Set as Set

type Point = (Word32, Word32)

makeTestTriple :: Float -> Float -> Float -> Float -> [(Float, Float)] -> (Point, Point,
    Set.Set Point)
makeTestTriple x1 y1 x2 y2 trav = (makePoint x1 y1, makePoint x2 y2, Set.fromList (map
    (\(a,b) -> makePoint a b) trav))

makePoint :: Float -> Float -> Point
makePoint x y = (floatToWord x, floatToWord y)

makeTestSet :: Int -> Set.Set Point
makeTestSet n = Set.fromList [ makePoint (fromIntegral a) ((fromIntegral a) + 1) | a <-
    [1..n]]

convertFloat :: [Point] -> [(Float, Float)]
convertFloat pts = [(wordToFloat a, wordToFloat b) | (a, b) <- pts]

getDistance :: Point -> Point -> Word32
getDistance (x1, y1) (x2, y2) = floatToWord ((((wordToFloat x1 - wordToFloat x2) ** (2 ::
    Float)) + ((wordToFloat y1 - wordToFloat y2) ** (2 :: Float))) ** (0.5 :: Float))

getPathDistance :: Set.Set Point -> Word32
getPathDistance pts | (Set.size pts) < 2 = floatToWord 0.0
                    | otherwise = floatToWord (sum (map wordToFloat (zipWith
```

```
    (getDistance) pt_lst pt_tail)))
                                    where
                                        pt_lst = Set.toList pts
                                        pt_tail = tail pt_lst


triples :: Point -> [Point] -> [(Point, Point, [Point])]
triples start_pt points = [(start, end, trav) | start <- points, end <- points, start ==
    start_pt, start /= end, let trav = List.delete start $ List.delete end points]

permShort :: [Point] -> [[Point]]
permShort xs = [ x : [y | y <- xs, y /= x] | x <- xs ] --`using` parList rpar

minTrip :: (Point, Point, Set.Set Point) -> Word32
minTrip (start_pt, end_pt, pts) | Set.size pts == 0 = getDistance start_pt end_pt
                                | Set.size pts == 1 = floatToWord (wordToFloat
    (getDistance start_pt (Set.elemAt 0 pts)) + wordToFloat (getDistance (Set.elemAt 0
    pts) end_pt))
                                | otherwise = floatToWord (wordToFloat (getDistance
    end_pt first_elem) + wordToFloat (minimum (map minTrip [(start_pt, first_elem,
    Set.fromList ps) | ps <- pms])))
                                                where
                                                    first_elem = Set.elemAt 0 pts
                                                    pms = permShort . Set.toList . snd $
    (Set.splitAt 1 pts)


minTripFib :: (Point, Point, Set.Set Point) -> Word32
minTripFib = memoize $ \(start_pt, end_pt, pts) -> case Set.size pts of
        0 -> getDistance start_pt end_pt
        1 -> floatToWord (wordToFloat (getDistance start_pt first_elem) + wordToFloat
    (getDistance first_elem end_pt))
            where
                first_elem = Set.elemAt 0 pts
        _ -> floatToWord (wordToFloat (getDistance end_pt first_elem) + wordToFloat
    (minimum (map minTripFib [(start_pt, first_elem, Set.fromList ps) | ps <- pms])))
    --parmap rpar
            where
                first_elem = Set.elemAt 0 pts
                pms = permShort . Set.toList . snd $ (Set.splitAt 1 pts)
```

HeldKarpSetPar.hs: (minTripFib Contains the memoization)

```
module HeldKarpSet_Par
(makeTestTriple,
makePoint,
makeTestSet,
convertFloat,
getDistance,
getPathDistance,
triples,
permShort,
minTrip,
minTripFib)
where
import qualified Data.List as List
```

```
import Data.Function.FastMemo
import Data.Word
import Data.FloatCast
import qualified Data.Set as Set
import Control.Parallel.Strategies

type Point = (Word32, Word32)

makeTestTriple :: Float -> Float -> Float -> Float -> [(Float, Float)] -> (Point, Point,
    Set.Set Point)
makeTestTriple x1 y1 x2 y2 trav = (makePoint x1 y1, makePoint x2 y2, Set.fromList (map
    (\(a,b) -> makePoint a b) trav))

makePoint :: Float -> Float -> Point
makePoint x y = (floatToWord x, floatToWord y)

makeTestSet :: Int -> Set.Set Point
makeTestSet n = Set.fromList [ makePoint (fromIntegral a) ((fromIntegral a) + 1) | a <-
    [1..n]]

convertFloat :: [Point] -> [(Float, Float)]
convertFloat pts = [(wordToFloat a, wordToFloat b) | (a, b) <- pts]

getDistance :: Point -> Point -> Word32
getDistance (x1, y1) (x2, y2) = floatToWord ((((wordToFloat x1 - wordToFloat x2) ** (2 ::
    Float)) + ((wordToFloat y1 - wordToFloat y2) ** (2 :: Float))) ** (0.5 :: Float))

getPathDistance :: Set.Set Point -> Word32
getPathDistance pts | (Set.size pts) < 2 = floatToWord 0.0
                    | otherwise = floatToWord (sum (map wordToFloat (zipWith
    (getDistance) pt_lst pt_tail)))
                                    where
                                        pt_lst = Set.toList pts
                                        pt_tail = tail pt_lst


triples :: Point -> [Point] -> [(Point, Point, [Point])]
triples start_pt points = [(start, end, trav) | start <- points, end <- points, start ==
    start_pt, start /= end, let trav = List.delete start $ List.delete end points]

permShort :: [Point] -> [[Point]]
permShort xs = [ x : [y | y <- xs, y /= x] | x <- xs ] `using` parList rpar

minTrip :: (Point, Point, Set.Set Point) -> Word32
minTrip (start_pt, end_pt, pts) | Set.size pts == 0 = getDistance start_pt end_pt
                                | Set.size pts == 1 = floatToWord (wordToFloat
    (getDistance start_pt (Set.elemAt 0 pts)) + wordToFloat (getDistance (Set.elemAt 0
    pts) end_pt))
                                | otherwise = floatToWord (wordToFloat (getDistance
    end_pt first_elem) + wordToFloat (minimum (map minTrip [(start_pt, first_elem,
    Set.fromList ps) | ps <- pms])))
                                                    where
                                                        first_elem = Set.elemAt 0 pts
                                                        pms = permShort . Set.toList . snd $
    (Set.splitAt 1 pts)
```

```
minTripFib :: (Point, Point, Set.Set Point) -> Word32
minTripFib = memoize $ \(start_pt, end_pt, pts) -> case Set.size pts of
        0 -> getDistance start_pt end_pt
        1 -> floatToWord (wordToFloat (getDistance start_pt first_elem) + wordToFloat
    (getDistance first_elem end_pt))
            where
                first_elem = Set.elemAt 0 pts
        _ -> floatToWord (wordToFloat (getDistance end_pt first_elem) + wordToFloat
    (minimum (parMap rpar minTripFib [(start_pt, first_elem, Set.fromList ps) | ps <-
    pms]))) --parmap rpar
            where
                first_elem = Set.elemAt 0 pts
                pms = permShort . Set.toList . snd $ (Set.splitAt 1 pts)
```