AStar: Parallel Functional Programming Final Project 2022
Donghan Kim (dk3245)

Introduction:

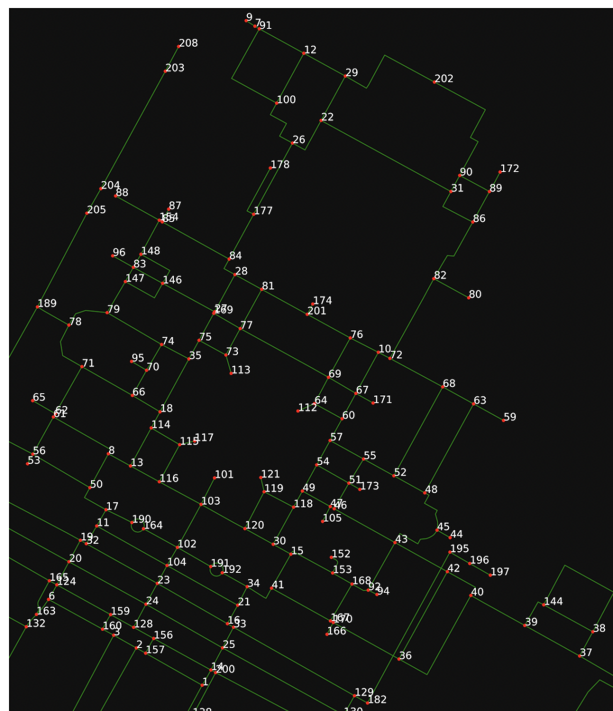
Finding the shortest path between two points on a graph is a popular computer science problem even researched today. While there are many complete shortest-path algorithms, people often look for one that is not only complete but also fast. Improved upon the work of Dijkstra's algorithm, the A* search algorithm incorporates a heuristic function to the existing cost function to allow the search to occur in a more targeted manner. My objective was to create a sequential and parallel version of the A* search, and explore what functional programming methods can be used to decrease the time of A* search.

Open Street Maps:

To properly test my A* search algorithm, I used real-world data from OpenStreetMaps (OSM). OSM is a popular and well-respected open-source dataset that can convert road-like infrastructure into any graph configuration, whether weighted/unweighted directed or undirected. To properly utilize the OSM dataset, I used a Python package called OSMnx that allowed me to extract graphical information pertaining to certain areas in New York City. The three graphs I used to test my algorithm were: Columbia University (210 nodes), Central Park (1397 nodes), and Manhattan (43404 nodes).

Graph Representation:

All three graphs are represented in the same way. They are weighted, bi-directional graphs where each edge represents a walkable path in the location that the graph is representing. To run A* search on this graph, I decided to export the graph data into a .txt file that can be loaded into my Haskell program. Each line in the .txt file contains information about a single node in the graph. The line is separated by delimiters that I chose to make it easier to load the .txt file into my Haskell program. All nodes have an index (idx) number that is used to reference the node it belongs to in the OSM dataset, longitude and latitude coordinates (7-point precision), and a list of tuples representing its adjacent nodes with its respective edge weights (similar to adjacency list). The edge weights are provided by OSM, and denote how long (in seconds) it would take to reach one node to another, assuming a speed of 5.1km/h. In Haskell, I



created a datatype called Node to contain this information and an Data.IntMap stores the node information in an efficient data structure for efficient look-up.

```haskell
-- | Node datatype
-- idx: Node index (used as key in nodeMap)
-- coord: lat,lon coordinates
-- edges: list of (idx,gn) tuples
data Node = Node { idx   :: Int,
                   coord :: (Double, Double),
                   edges :: [(Int, Double)]
                   } deriving (Show)
```

A* Search:

As mentioned above, A* search differs from Dijkstra's algorithm in that exploration (search) is targeted. More specifically, in A* search, we use a heuristic function that decides which nodes should be explored first. Of course, nodes with a high heuristic cost will most likely be explored last. Because of this distinction, while A* is a complete algorithm, it does not find the shortest path between all nodes in the graph (like Djikstra's algorithm). Instead, it only finds the shortest path between the source (starting) and target (destination) node.

How we select our heuristic function is extremely important, as it will determine how fast A* search can find its target. In addition to the heuristic function, we use the edge weights to represent the path cost, meaning the cost from the source node to the current node, g(n). Combining the path cost with the heuristic, we can calculate the total cost for a given node, commonly denoted as f(n).

$$f(n) = g(n) + h(n)$$

In the Haskell implementation, the total cost can be obtained using the calcFn function, which calculates the path cost and heuristic separately.

```haskell
-- | Calculate fn
calcFn :: Int -> Int -> Int -> M.IntMap Node -> M.IntMap Int -> Double
calcFn cIdx sIdx tIdx nodeMap cameFrom =
  let
    gn = calcGn cIdx sIdx nodeMap cameFrom 0.0
    hn = calcHn (nodeMap M.! cIdx) (nodeMap M.! tIdx)
  in gn + hn
```

Haversine Formula:

While the path cost is given, the heuristic is not. The better the heuristic, the better A* can be. In my implementation, I decided to utilize the longitude and latitude coordinates of the nodes, and estimate the physical distance between them as it is directly correlated to the amount of time it would take to reach the node. Many well-known formulas can calculate the distance between two lat/lon pairs. The Haversine formula is very popular due to its low computation to accuracy ratio.

$$
\begin{aligned}
d &= 2r \arcsin\left(\sqrt{\operatorname{hav}(\varphi_2 - \varphi_1) + (1 - \operatorname{hav}(\varphi_1 - \varphi_2) - \operatorname{hav}(\varphi_1 + \varphi_2)) \cdot \operatorname{hav}(\lambda_2 - \lambda_1)}\right) \\
&= 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \left(1 - \sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) - \sin^2\left(\frac{\varphi_2 + \varphi_1}{2}\right)\right) \cdot \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right) \\
&= 2r \arcsin\left(\sqrt{\sin^2\left(\frac{\varphi_2 - \varphi_1}{2}\right) + \cos\varphi_1 \cdot \cos\varphi_2 \cdot \sin^2\left(\frac{\lambda_2 - \lambda_1}{2}\right)}\right).
\end{aligned}
$$

Although I my A* search does not use the Haversine formula, it is available to use in the src/Lib.hs file.

Vincenty Formula:

From my research, the Vincenty formula is the most accurate method for calculating the distance between two lat/long coordinates. There are several ways we can utilize Vincencty's formula to calculate the distance, but I decided to implement the inverse method to estimate distance the distance.

$$
\begin{aligned}
\sin\sigma &= \sqrt{(\cos U_2 \sin\lambda)^2 + (\cos U_1 \sin U_2 - \sin U_1 \cos U_2 \cos\lambda)^2} \\
\cos\sigma &= \sin U_1 \sin U_2 + \cos U_1 \cos U_2 \cos\lambda \\
\sigma &= \operatorname{arctan2}(\sin\sigma, \cos\sigma)^{[1]} \\
\sin\alpha &= \frac{\cos U_1 \cos U_2 \sin\lambda}{\sin\sigma}{}^{[2]} \\
\cos(2\sigma_{\mathrm{m}}) &= \cos\sigma - \frac{2\sin U_1 \sin U_2}{\cos^2\alpha} = \cos\sigma - \frac{2\sin U_1 \sin U_2}{1 - \sin^2\alpha}{}^{[3]} \\
C &= \frac{f}{16}\cos^2\alpha\left[4 + f\left(4 - 3\cos^2\alpha\right)\right] \\
\lambda &= L + (1 - C) f \sin\alpha \left\{\sigma + C\sin\sigma\left[\cos(2\sigma_{\mathrm{m}}) + C\cos\sigma\left(-1 + 2\cos^2(2\sigma_{\mathrm{m}})\right)\right]\right\}
\end{aligned}
$$

Here we iterate and evaluate lambda, until either a max iteration is met, or lambda converges. Once the iterations are complete, we can utilize the values to estimate our distance:

$$u^2 = \cos^2\alpha\left(\frac{a^2 - b^2}{b^2}\right)$$

$$A = 1 + \frac{u^2}{16384}\left(4096 + u^2\left[-768 + u^2\left(320 - 175u^2\right)\right]\right)$$

$$B = \frac{u^2}{1024}\left(256 + u^2\left[-128 + u^2\left(74 - 47u^2\right)\right]\right)$$

$$\Delta\sigma = B\sin\sigma\left\{\cos(2\sigma_{\mathrm{m}}) + \frac{1}{4}B\left(\cos\sigma\left[-1 + 2\cos^2(2\sigma_{\mathrm{m}})\right] - \frac{B}{6}\cos[2\sigma_{\mathrm{m}}]\left[-3 + 4\sin^2\sigma\right]\left[-3 + 4\cos^2(2\sigma_{\mathrm{m}})\right]\right)\right\}$$

$$s = bA(\sigma - \Delta\sigma)$$

S here represents the distance in meters. If the convergence tolerance is set to 1e-12, Vincenty's formula guarantees margin error to be less than 0.06mm. The current iteration I have uploaded with this report uses the Vincecy formula to assign the heuristic (called from calcHn).

```haskell
-- | Vincenty forumla (WGS-84 standard, in meters)
-- accurate to upto 0.066mm, but expensive
vincenty :: (Double, Double) -> (Double, Double) -> Double
vincenty n1 n2 =
  let lon1 = fst n1
      lat1 = snd n1
      lon2 = fst n2
      lat2 = snd n2
      a = 6378137 :: Double
      f = 1/298.257223563 :: Double
      b = (1 - f) * a :: Double
      capL = (lon2 - lon1) * pi / 180 :: Double
      u1 = atan ((1 - f) * tan (lat1 * pi / 180)) :: Double
      u2 = atan ((1 - f) * tan (lat2 * pi / 180)) :: Double
      sinU1 = sin u1 :: Double
      cosU1 = cos u1 :: Double
      sinU2 = sin u2 :: Double
      cosU2 = cos u2 :: Double

  in iterateUntilClose 1e-12 10000 (\lambda ->
    let
        sinSigma = sqrt ((cosU2 * sin (lambda))**2 + (cosU1 * sinU2 - sinU1 * cosU2 * cos (lambda))**2) :: Double
        cosSigma = sinU1 * sinU2 + cosU1 * cosU2 * cos lambda :: Double
        sigma = atan (sinSigma/cosSigma) :: Double
        sinAlpha = cosU1 * cosU2 * (sin lambda) / sinSigma :: Double
        cosSqAlpha = 1 - sinAlpha**2 :: Double
        cos2sig = cosSigma - (2*sinU1*sinU2/cosSqAlpha) :: Double
        capC = (f/16)*cosSqAlpha*(4 + f*(4 - 3*cosSqAlpha)) :: Double
        lambda' = capL + (1 - capC) * f * sinAlpha * (sigma + capC * sinSigma * (cos2sig + capC * cosSigma * (-1 + 2 * cos2sig**2))) :: Double
    in (lambda, lambda', sigma, cosSqAlpha, cosSigma, cos2sig, sinSigma, a, b)) capL
  where
    iterateUntilClose tolerance maxIterations f lambda =
      let (prevLambda, newLambda, sig, csa, cs, c2s, ss, defA, defB) = f lambda
      in if abs (prevLambda - newLambda) < tolerance || maxIterations == 0
         then vincentyDistance ((csa**2)*(defA**2 - defB**2)/defB**2) ss c2s cs defB sig
         else iterateUntilClose tolerance (maxIterations - 1) f newLambda
```

Sequential Implementation:

In addition to the graph data, node map, and cost function, we need several more data structures to complete A* search. In particular, we need a priority queue that can sort the nodes to explore based on their total cost. This priority queue is referred to as the openList and uses the Data.MinPQueue data structure in Haskell. To minimize the memory burden, I decided to store a tuple where the first element is the total cost (Double) and the second element is the index of the node (Int). This is reasonable since the node map grants constant look-up performance.

```haskell
-- | Update openList
updateOpen :: P.MinQueue (Double, Int) -> [(Int, Double)] -> (P.MinQueue (Double, Int), P.MinQueue (Double, Int))
updateOpen openList fcost =
  let
    currentNodes = map snd (P.elemsU openList)
    newNodes = [(fn, idx) | (idx, fn) <- fcost, idx `notElem` currentNodes]
    newOpen = P.fromList newNodes

    temp = P.partition (\(fn, idx) -> let fn' = lookup idx fcost in (isJust fn' && fromJust fn' < fn)) openList
    sameOpen = snd temp
    updatedOpen = P.map (\(_, idx) -> (fromJust $ lookup idx fcost, idx)) (fst temp)
  in if P.null openList then (P.union newOpen updatedOpen, sameOpen) else (P.union newOpen updatedOpen, sameOpen)
```

The openList is constantly updated to reflect newly calculated costs, and to add additional nodes to explore. Furthermore, we also need to store where nodes “came from”. This is extremely important since the total cost of a given node can change. This is similar to Dijkstra’s algorithm where the “better” or cheaper path should result in updating the cost of the node.

```haskell
-- | Update cameFrom
updateFrom :: M.IntMap Int -> [(Int, Int)] -> M.IntMap Int
updateFrom cameFrom adjNodes = foldl f cameFrom adjNodes
  where
    f cameFrom (from,to) = if M.member from cameFrom then cameFrom else M.insert from to cameFrom
```

Lastly, we also maintain a closed list, which contains all the nodes that have been explored, preventing an infinite loop. In Haskell, I used the Data.IntSet data structure to represent the closed list, and therefore named it “closedSet”.

Parallel Implementation:

The two largest bottlenecks of the A* search algorithm are the I/O (reading the graph data) and in my case, the heuristic calculation. Fortunately, the heuristic calculation is mutually exclusive, meaning they can be calculated in parallel. However, to truly see the advantage of parallelism, the average out-degree (average number of adjacent nodes) has to be fairly high; otherwise, the cost of creating parallel computations will outweigh the advantage. I created two separate parallel functions using the to decrease the running time of my algorithm.

```haskell
-- | parallel calculation
parCalcFn :: Int -> Int -> Int -> M.IntMap Node -> M.IntMap Int -> Double
parCalcFn cIdx sIdx tIdx nodeMap cameFrom =
  let (hn, gn) = runEval $ do {
    ; hn <- rpar $ calcHn (nodeMap M.! cIdx) (nodeMap M.! tIdx)
    ; gn <- rpar $ calcGn cIdx sIdx nodeMap cameFrom 0.0
    ; return (hn, gn)
  }
  in gn + hn
```

```haskell
-- calculate f(n) = g(n) + h(n)
adjNodes = filter ((\adjIdx -> S.notMember adjIdx closedSet').fst) (edges cNode)
cameFrom' = updateFrom cameFrom [(adjIdx, cIdx) | (adjIdx, _) <- adjNodes]
fcost = parMap rdeepseq (\(adjIdx, _) -> (adjIdx, calcFn adjIdx sIdx tIdx nodeMap cameFrom')) adjNodes
```

Similar to the sequential implementation, the parCalcFn calculates the path cost and heuristic cost in parallel. Of course, the heuristic calculation is much more expensive, however, with more large enough graphs, there can be some added benefits. In addition, I used parMap to calculate the total cost, f(n) of all nodes in the openList in parallel. As mentioned above, there only seem to be any benefits when run on graphs that are magnitudes larger than the one I tested, however, for large enough graphs this can make a huge difference. Unfortunately, I could not figure out how to parallelize reading the graph data, finding the right “chunk” size to seek into different parts of the file was more challenging than I initially thought.

<u>Project Structure:</u>

```
.
├── LICENSE
├── README.md
├── astar
│   ├── CHANGELOG.md
│   ├── Setup.hs
│   ├── astar.cabal
│   ├── data (not included)
│   │   ├── CentralPark.graphml
│   │   ├── CentralPark.pickle
│   │   ├── CentralPark.png
│   │   ├── CentralPark.txt
│   │   ├── ColumbiaUniversity.graphml
│   │   ├── ColumbiaUniversity.pickle
│   │   ├── ColumbiaUniversity.png
│   │   ├── ColumbiaUniversity.txt
│   │   ├── Manhattan.graphml
│   │   ├── Manhattan.pickle
│   │   ├── Manhattan.png
│   │   └── Manhattan.txt
│   ├── package.yaml
│   ├── parApp
│   │   └── Main.hs
│   ├── seqApp
│   │   └── Main.hs
│   ├── src
│   │   └── Lib.hs
│   ├── stack.yaml
│   ├── stack.yaml.lock
│   └── test
│       └── Spec.hs
├── demo_seq.sh
├── graph.py
└── requirements.txt
```

The atar/ folder contains the Haskell project created using Stack. Furthermore, the graph.py Python script contains all the code needed to generate, and run a plot of the shortest path calculated in Haskell. Both .hs executables (seq for sequential and par for parallel) create a .txt file called **res.txt** at the location when you run the Haskell application. You can load the res.txt file and plot the path using the python script.

Please refer to the installation details, and graph data generation details on my Github repository: https://github.com/donghankim/ParAStar