COMS4995 Report - A* Search For TSP

Adele Bai (ayb2121)
Vincent Mutolo (vm2724)

Introduction & Problem

The goal is to parallelize an A-star search implementation on the Traveling Salesman Problem.

- Traveling Salesman Problem - given a connected graph, find the shortest path that touches all the vertices and returns to the starting point.
- A-star search - this is basically dijkstra's shortest path algorithm but with a heuristic added to the cost. In our implementation we set the heuristic value to 0 (for simplicity) so we basically implemented dijkstra's algorithm.

Runs are benchmarked on the following machines:

```
Unset
Machine 1 (Adele's PC)
Architecture:            x86_64
  CPU op-mode(s):        32-bit, 64-bit
CPU(s):                  12
  On-line CPU(s) list:   0-11
Vendor ID:               GenuineIntel
  Model name:            Intel(R) Core(TM) i7-8700K CPU @ 3.70GHz

Machine 2 (DigitalOcean)
Architecture:              x86_64
  CPU op-mode(s):          32-bit, 64-bit
CPU(s):                    16
  On-line CPU(s) list:     0-15
Vendor ID:                 GenuineIntel
  BIOS Vendor ID:          QEMU
  Model name:              Intel(R) Xeon(R) Platinum 8168 CPU @ 2.70GHz
       BIOS Model name:    pc-i440fx-6.1  CPU @ 2.0GHz
       Model:              85
       Thread(s) per core:   1
```

Datasets

We used fully connected graphs sourced from a collection of common TSP problems with known solutions. A small 5-city set (5! = 120 permutations) was used to quickly verify the implementation's correctness.

The main dataset for assessing performance is a fully connected graph with 17 cities with a minimum cost of 2085. A 17 city TSP problem has 17! = 355 trillion permutations so a brute-force solution would take 41 days (assuming 10^8 iterations per second).

This problem size is sufficiently large to keep the execution CPU heavy without trading off the velocity of our working process (i.e. our serial A* implementation took around 25 seconds). The input file (/data/17_cities_edges.csv) is trivially small so there is no IO bottleneck when parsing the graph itself.

A* Serial Implementation

We first implemented A* search to solve the Traveling Salesman Problem (TSP) serially. TSP involves finding the shortest possible route that visits each city in a given list exactly once and returns to the starting city. The algorithm begins by selecting an arbitrary starting city and initializes the search frontier with a node representing this starting point. The goal is defined as a state where all cities have been visited, forming a complete route that returns to the start.

Crucial here is how we define our states (i.e. vertices in the search, which are distinct from the cities in TSP). A state consists of the tuple of the current city of the traveler and the set of cities they have visited. Defining a state this way lets us avoid wasted work in re-exploring essentially equivalent states. The algorithm keeps track of a global starting node, so any two states that consist of the same set of already-visited cities (and the same current city) are effectively equivalent because the only decisions are how to visit the rest of the cities from the current one.

```
data Node = Node {
    city :: City,
    path :: [City],
    gCost :: Distance,  -- Cost so far
    fCost :: Distance   -- (gCost + heuristic)
} deriving (Show)
```

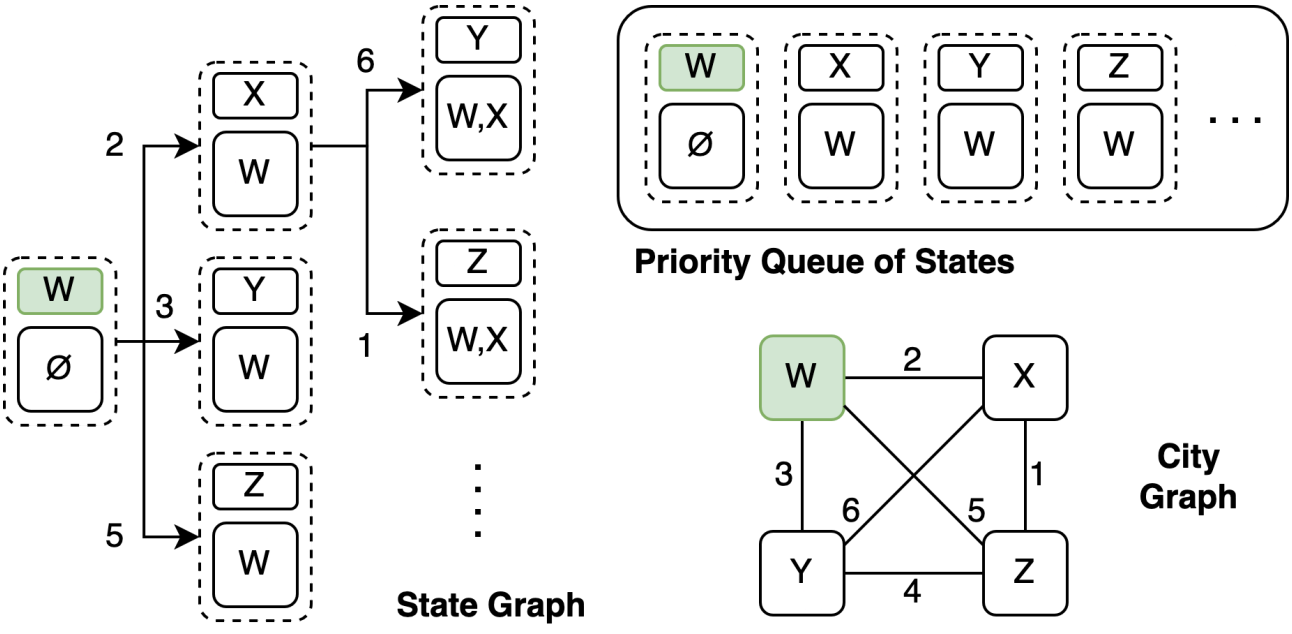
VisitedSet	
Set{2,4}	1
Set{7,8}	3
Set{1,2,5}	8

The search operates by exploring paths in the order of their estimated total cost, which is the sum of the cost to reach the current node and a heuristic estimate of the cost to reach the goal

from that node. The heuristic can be either a constant zero for simplicity or something more complex like an MST-based estimate for TSP. The only hard requirement is that the heuristic be "admissible", meaning it never overestimates the cost. Again, each node in the search represents a partial route, including the current city, the path taken so far, and the total cost up to that point. The heuristic function guides the search by providing an optimistic estimate of the remaining cost, helping to prioritize nodes that are likely to lead to the optimal solution.

As the algorithm progresses, it expands nodes by generating successor states, i.e. possible next cities to visit that haven't been visited yet. It adds these successors to the frontier (priority queue), where they are ordered based on their estimated total cost. The algorithm keeps track of visited states to avoid revisiting the same configurations and to prevent cycles. It continues expanding nodes and exploring new paths until it finds a node that reaches the goal state, meaning all cities have been visited and the path returns to the starting city. At this point, the algorithm returns the optimal route found, which represents the shortest possible tour covering all cities exactly once.

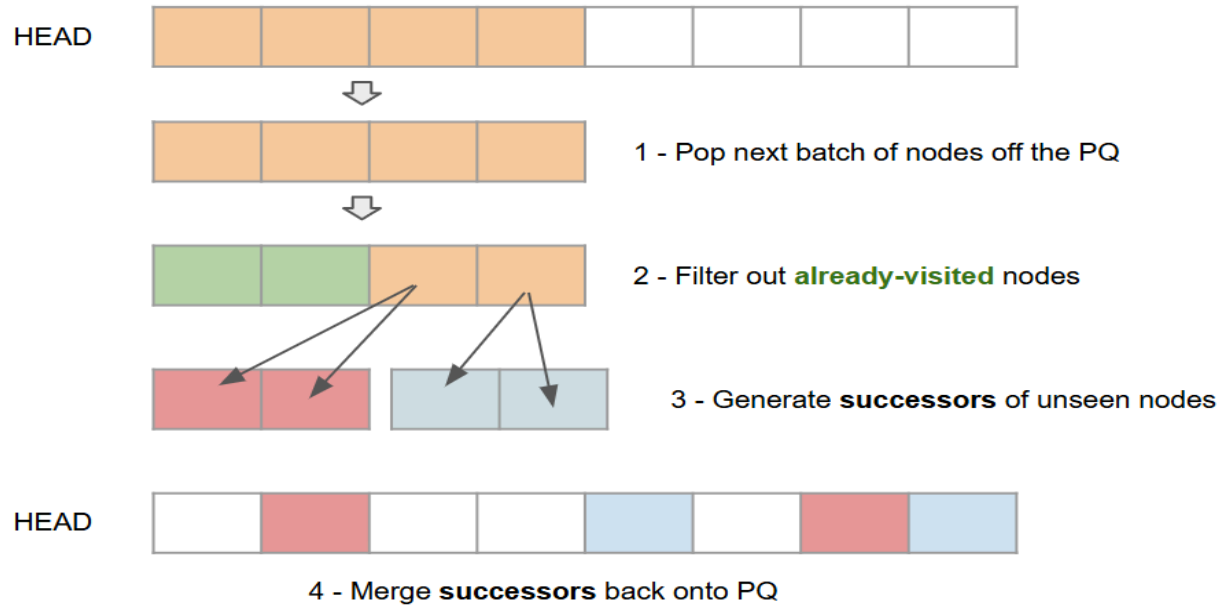
The below diagram illustrates how a city graph turns into a logical graph over “states” of TSP representing what cities have already been visited and how they’re arranged by cost in the priority queue to visit next. The diagram below assumes a constant heuristic of 0, an arbitrary starting point of W, and that only the successors of W have been added to the queue. The main point is to illustrate the structure of how states are stored as a tuple of current city and previous cities (and current cost so it doesn’t have to be recomputed all the time).



Parallelization Approach & Challenges

The goal of the parallelization is to speed up the work processing for the same-sized problem rather than speed up the processing of multiple problems. The main approach we explored that gave promising results was processing nodes off the priority queue in parallel. This method stays true to the spirit of A* by continuing to prioritize good nodes. In the serial approach, only the head of the queue is explored at a time (to depth 1). In the parallel approach, we explore up to **k** nodes off the queue in a batch. This was done with the `Control.Parallel.Strategies` library.

See below for a visualization of an iteration under this approach.



The main implementation challenges and features of this approach are outlined below.

Ensuring correctness of the visited states collection

In the serial implementation it was trivial to maintain a set of visited states. To check whether a node had already been visited - you just check its membership in the set. After it has been explored, it is added to the visited set. This works because when only a single item is explored at once, it is guaranteed that nothing will generate a better version of its equivalent ‘state’ and we can discard all future equivalent encounters. This is **no longer true if multiple nodes are explored at the same time**.

In the parallel version, a node near the front of the queue could generate equivalent states of other (further) nodes in the same batch. In order to keep the visited set correct, we had to use a **Data.HashMap** to track the best cost of equivalent states. A node is only considered visited if

1. It has a cost better than the equivalent entry in the map or
2. It does not exist in the map.

Luckily, the performance impact of using a HashMap vs HashSet was negligible since they use the same underlying data structure.

Forcing deep parallel evaluation using deepseq

In the figure x above, step 2 and 3 are the easiest to parallelize since both involve an `fmap` operation on a list that could be independently executed across list entries. We played around different strategies (`rpar`, `rseq`, `rdeepseq`) but ultimately had to use `rdeepseq` to force the evaluation of the full structures because the shallow WHNF evaluation of `rpar`/`rseq` wasn't sufficient.

```
Unset
instance NFData Node where
    rnf (Node c p g f) = rnf c `seq` rnf p `seq` rnf g `seq` rnf f
```

Step 2 above (checking HashMap membership) was a large chunk of the execution (see a-star-tsp-haskell.exe.prof) and only full evaluation of this operation in parallel allows the program to scale. The main reason for this is that our implementation relies on immediate results to be available in order to resync them back into the priority-queue **before the next iteration can start**, rather than some recursive approach that lets us rely on the default lazy WNHF evaluation.

Controlling the number of sparks

Spinning off millions of short-lived sparks for every node in a batch initially showed weak scaling and many sparks being fizzled or garbage collected. To control the amount of work per spark, we used two mechanisms:

1. **Batch size per iteration** - this was the number of nodes to pop off the queue at once.
2. **ParListChunk** - this is a list strategy that divides a list into blocks of size m and evaluates the blocks in parallel. Using this in combination with (1) was critical in allowing more work to be done per spark.

The spark distribution before optimization with ParListChunk is as follows. All the work does end up converted, but the execution time is 5x slower than if they were chunked. This was with a batch size of 4000, chunk size of 1.

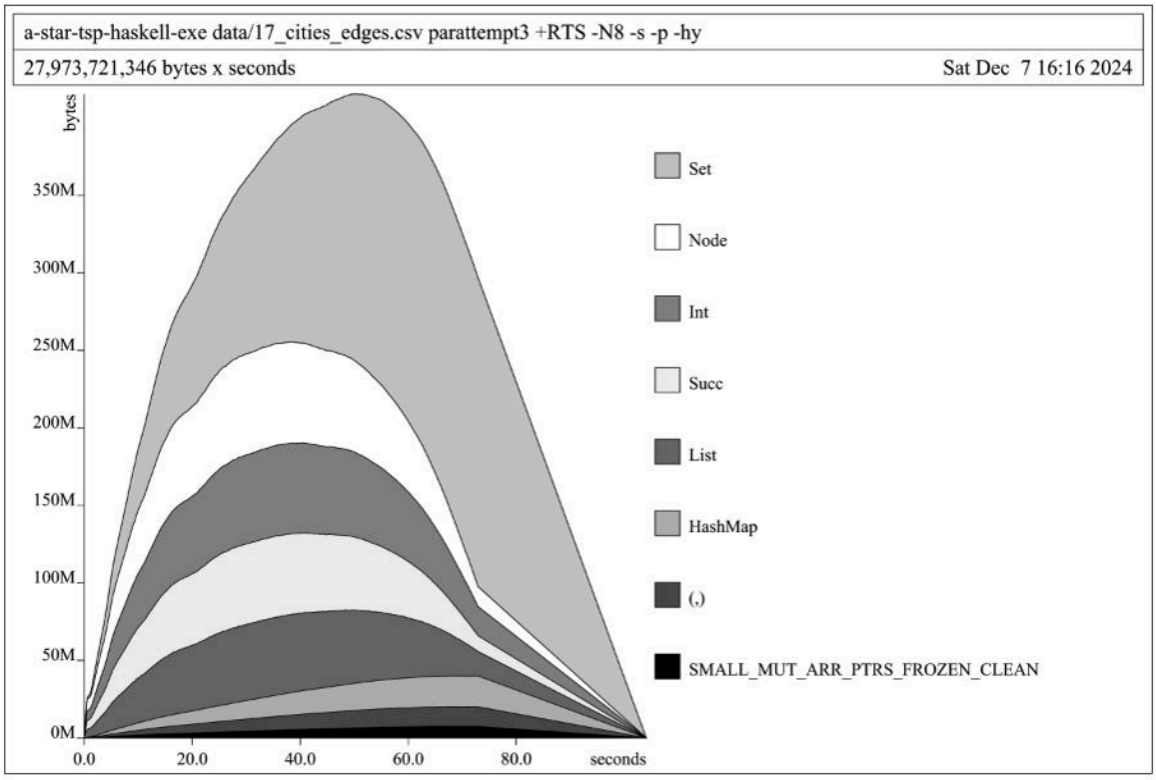
```
Unset
SPARKS: 5840192 (5631904 converted, 0 overflowed, 0 dud, 1713 GC'd, 206575
fizzled)
```

The optimal number of sparks for our benchmarks was somewhere between 20k-50k, with a batch size of 2400, and chunk size of 200:

```
Unset
SPARKS: 31748 (26665 converted, 0 overflowed, 0 dud, 46 GC'd, 5037 fizzled)
```

Mitigating GC throughput

The final challenge we discovered was the large (30%) of time spent on garbage collection when multiple cores (-N) were specified. Unfortunately we could not find an implementation solution for this as all the usage in the heap allocation map seemed organic - i.e. we know that nodes and sets (which are the key of the HashMap) make up the majority of memory allocation.



The memory churn resulted in about only 40% parallel GC work balance:

```
Unset
stack run 'data/17_cities_edges.csv' 'parattempt3' -- +RTS -N8 -s
Parallel GC work balance: 42.46% (serial 0%, perfect 100%)
  INIT    time    0.009s  (  0.033s elapsed)
  MUT     time   32.948s  ( 16.350s elapsed)
  GC      time   17.466s  (  8.315s elapsed)
```

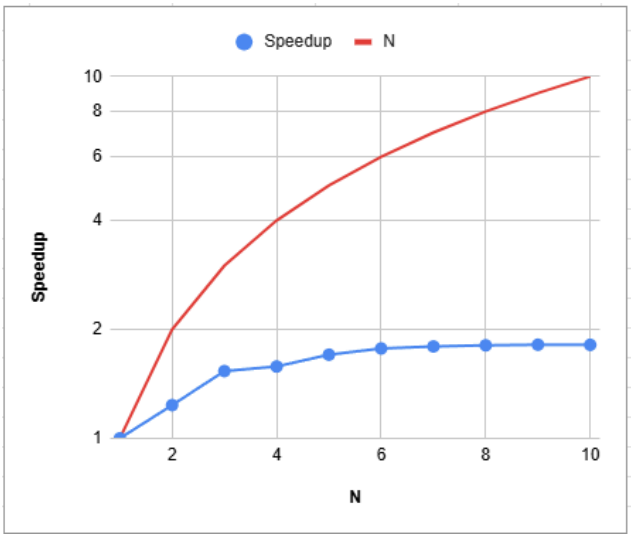
```
  EXIT    time    0.047s  (  0.005s elapsed)
  Total   time   50.469s  ( 24.703s elapsed)
```

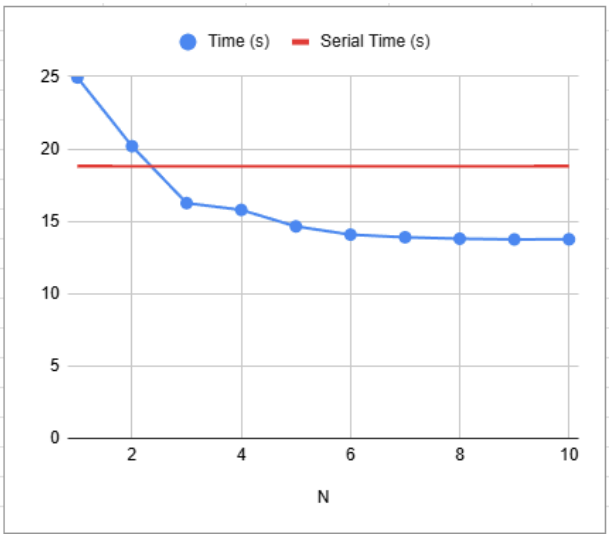
The way we mitigated this was by using an RTS flag to increase the GC allocation size (from the default 4MB to 32MB). This was done using the `-A32m` flag. I think this decreases the frequency of garbage collections.
For us, it reduced the GC time by 4x and increased GC efficiency to 95%.

```
Unset
stack run 'data/17_cities_edges.csv' 'parattempt3' -- +RTS -N8 -s -A32m
Parallel GC work balance: 95.13% (serial 0%, perfect 100%)
  INIT    time    0.007s  (  0.028s elapsed)
  MUT     time   33.093s  ( 16.267s elapsed)
  GC      time   13.885s  (  2.162s elapsed)
  EXIT    time    0.044s  (  0.007s elapsed)
  Total   time   47.029s  ( 18.464s elapsed)
```

Results - Generating successors of depth 1.

Note - these results were generated using **heuristic = 0 on Machine 1**.





We observed some scaling using the approach above and exploring every node to depth=1.
The scaling tapered off at around N=7.

N	Time (s)	Speedup
1	24.933	1
2	20.184	1.235
3	16.253	1.534
4	15.783	1.580
5	14.643	1.703
6	14.074	1.772
7	13.893	1.795
8	13.793	1.808
9	13.743	1.814
10	13.753	1.813

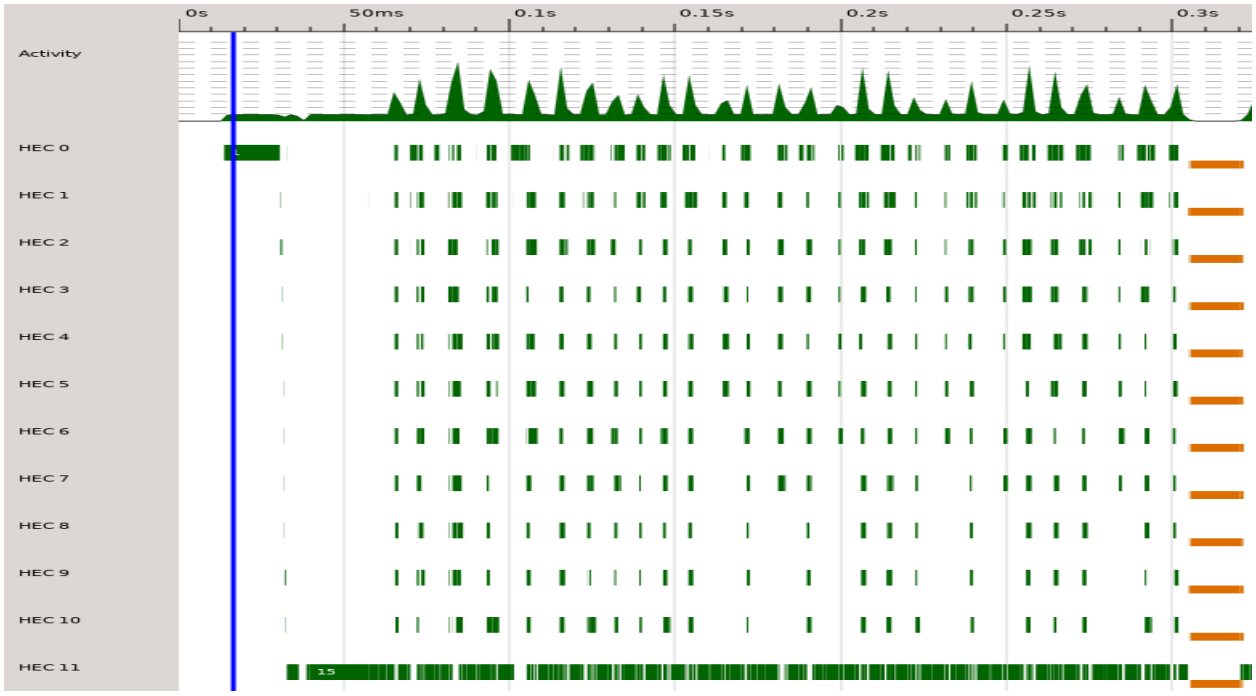
We could not get close to ideal scaling, but could beat the serial implementation (**18.75**s) at N=3.

The main problem with our approach is probably load balancing. With the batch size configuration of 2400 and chunks of 200, there 12 ‘blocking’ sparks every iteration so most workers are sitting idle, waiting on the slowest spark every iteration.

Summarized profiling rows show most of the bottleneck in the HashMap lookup (48.9%) and the priority queue rebalancing from popping multiple nodes off (splitAt, 25.6%).

```
Unset
Function, Type, % time (including inherited)
lookup#,  Data.HashMap.Internal, 48.9%
splitAt,  Data.PQueue.Min,  25.6%
hasVisitedBefore.element,  ParQueueProcessing,  14.7%  (this is allocating
elements of the HashMap)
misc 10%  (other stuff like rnf/rdeepseq)
```

A close up threadscope view shows 1) that sparks are **short-lived** (expected) resulting in spiky core usage and 2) potentially a lot of serial execution on a single core that could still be parallelized, but our guess is that core is actually doing the priority queue balancing, which can’t be parallelized.



Areas of improvement - could try to make each spark longer-lived and do more work. The risk here is the deeper the work, the higher the chance of it doing something not useful.

Solution - Adding a MST heuristic.

When applying A* to TSP, we have the option to specify a heuristic to guide the cost function A* uses in its priority queue. We started by leaving this heuristic a constant zero. This is valid since the only requirement on the heuristic is that it is "admissible", i.e. it never overestimates the cost of a solution.

However, we were searching for ways to scale better, and we thought that finding ways to distribute more individual work to sparks might help our scaling. That is, we wanted to find work that didn't require later synchronization. We figured that since we were parallelizing at the level of popping from the priority queue and searching, adding useful work to that search would give a better scaling ratio.

To that end, we implemented the Minimum Spanning Tree (MST) heuristic for A* search over TSP. So now the search considers the current city and calculates the cost to complete the tour by adding the cost from the current city to the nearest unvisited city (cost so far) and the cost of the MST covering all unvisited cities (estimate of remaining cost). This provides a bound on the remaining tour cost because the MST connects all unvisited cities with the minimal total edge weight without forming cycles. Since the MST cost underestimates or equals the minimal tour cost through the unvisited cities, the heuristic is admissible; it never overestimates the true minimal cost to reach the goal. This admissibility ensures that A* will find an optimal solution if one exists.

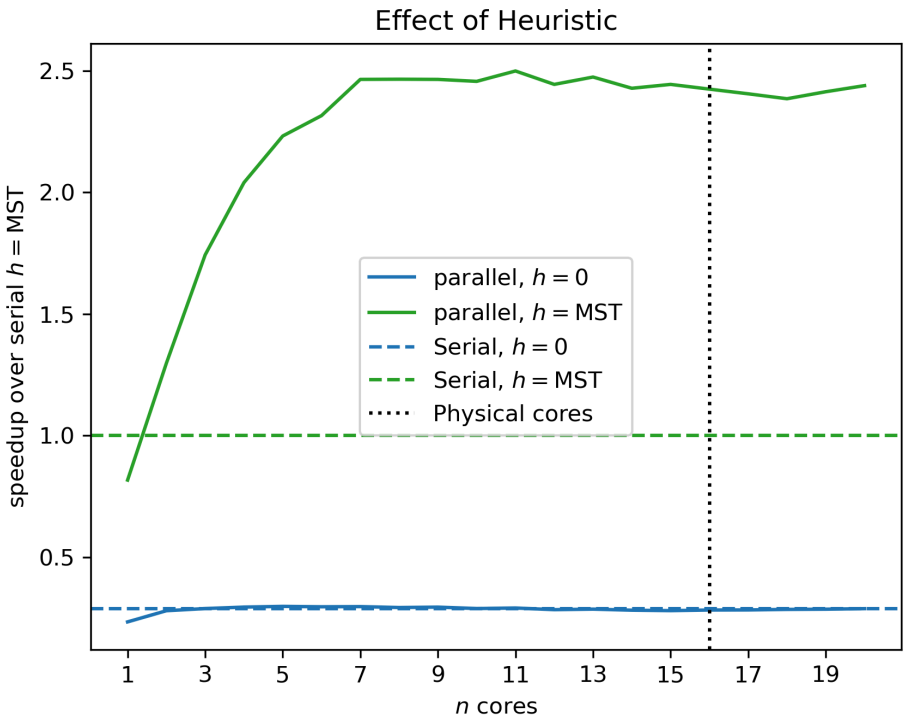


Figure to the left is benchmarking on Machine-2, showing minimal speedup from the heuristic. We now see decent linear scaling at about **0.5N from around N=1..5 cores**, and then the scaling gradually tapers off to constant, ending in an **overall 2.5x speedup over serial**. So the heuristic both improved the overall speed of the search (shown by the green

dotted line representing the serial implementation) and the parallel scaling properties.

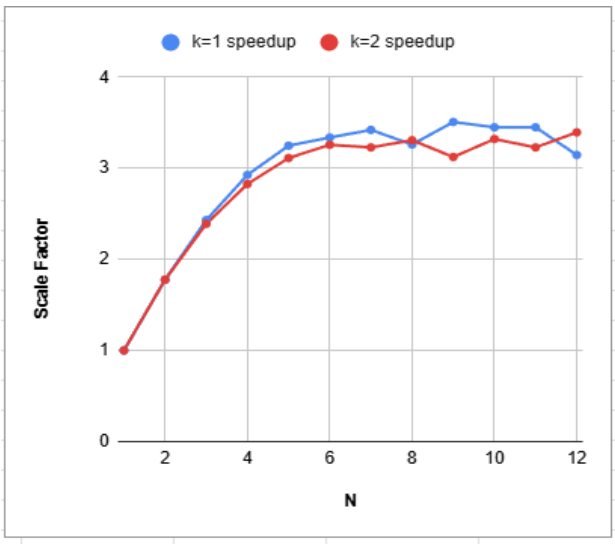
Results - Generating successors of depth k with a heuristic.

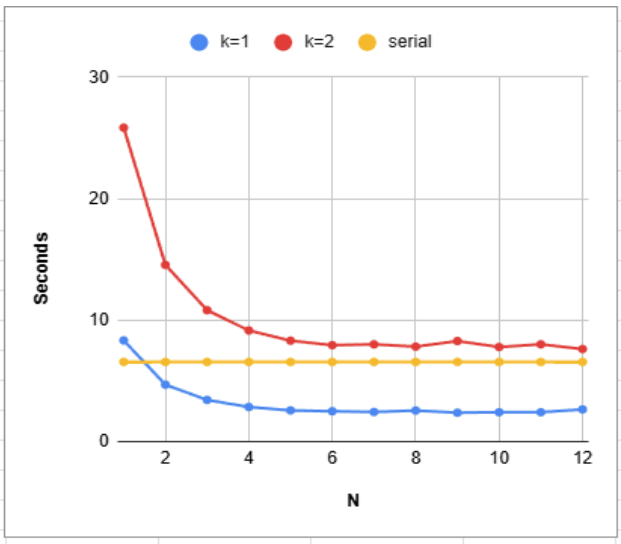
The idea behind this approach is to give each spark more work when it explores each node in parallel, to reduce the percentage of time spent checking the visited states membership. Adding the heuristic worked reasonably well in speeding up the program overall and reducing the number of membership checks.

Based on the profiling results below, a large % of time is now spent on computing the MST heuristic rather than doing HashMap lookups.

```
Unset
COST CENTRE                    MODULE   SRC                    %time %alloc
mstCost.sortedEdges            AStarLib.hs:81:5-77             42.3   36.7
kruskal                        AStarLib.hs:(97,1)-(105,40)     14.2    2.7
find                           AStarLib.hs:(109,1)-(113,33)    12.0    0.0
getEdgesBetween.collectEdges AStarLib.hs:(92,5)-(94,41)         8.1   15.2
getEdgesBetween.edges          AStarLib.hs:91:5-54              3.6   13.2
```

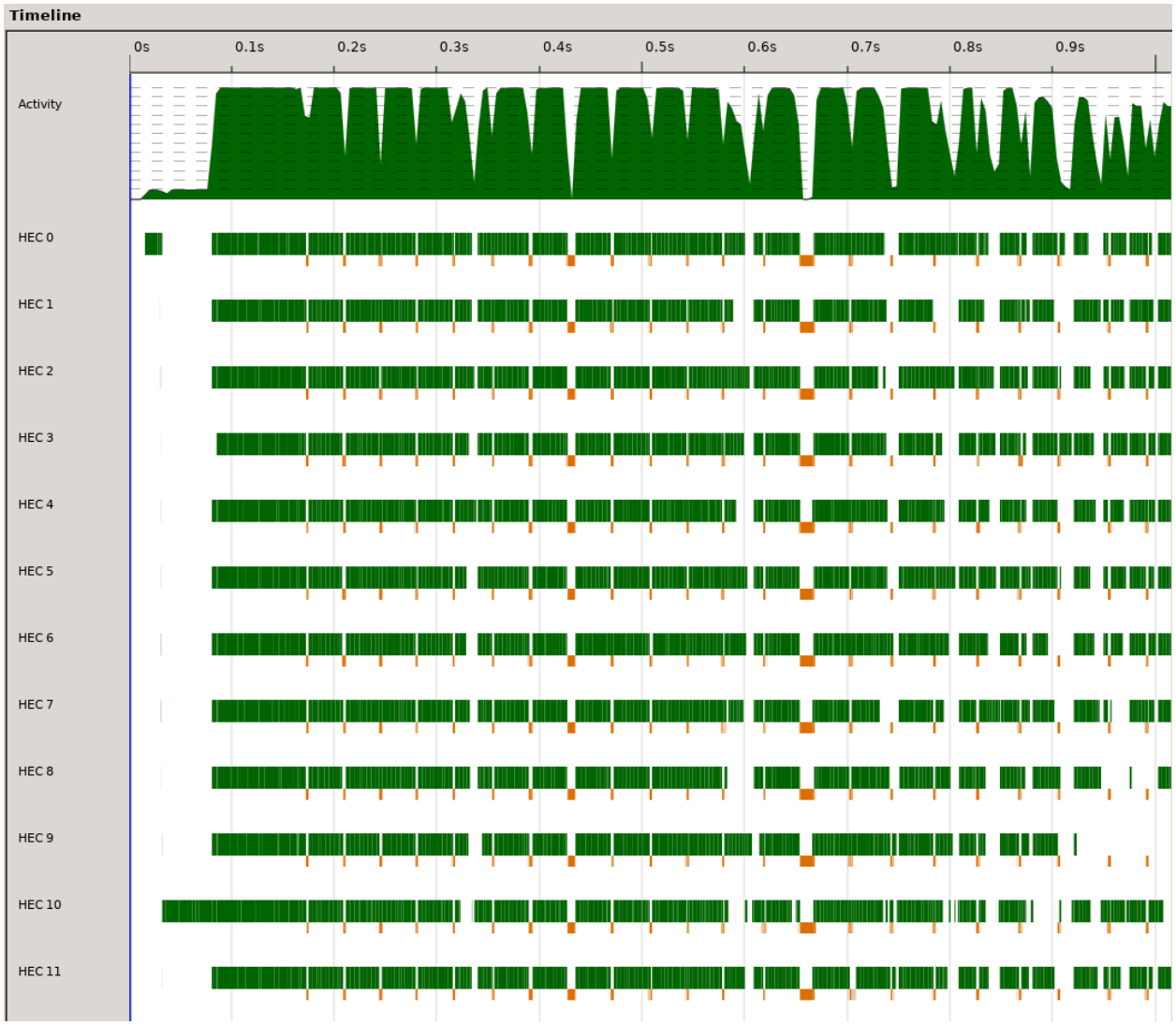
The batch size and chunk size had to be re-tuned to account for the higher workload per exploration step. The chunk size had to be reduced to be effective. A combination that worked well was `batch_size=600` and `chunk_size=10`.
Below shows benchmarking on **Machine-1** using a depth=1 and depth=2.





Unfortunately it seems that exploring more depth does not improve the overall runtime, but the result of the heuristic is extremely effective especially when compared to prior benchmarking. The serial execution took around 6.5s while the best parallel execution now takes 2.5s.

Threadscope also shows much better CPU utilization with more consistency (probably due to the lower chunk size, allowing for better load balancing). Even though there's still room for improvement (i.e. not all cores are in use 100% of the time), this looks much better than before.

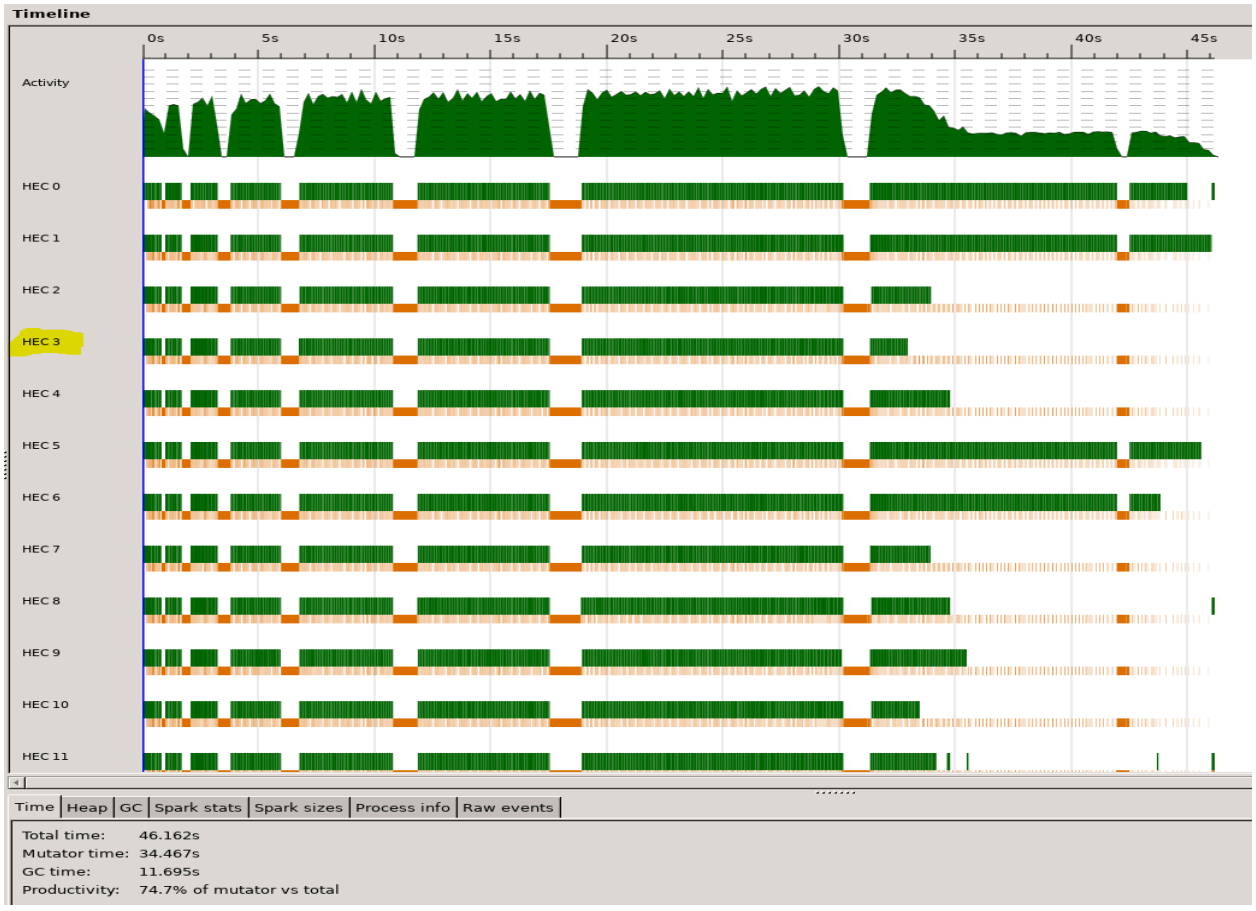


Other Unsuccessful Attempts (Naive Sharding)

One other approach we tried was naively sharding the search space to some depth k, and running the full exploration of each shard in parallel. The idea is that we do more work overall, but each shard has their exploration space reduced by a factor. For the 17 city problem this is expected to generate 17 sparks for depth=1, and 17*17 sparks for depth=2 and so on.

This approach scaled quite well, but in the end was still slower than the serial implementation. There are a few reasons this didn't work:

1. These are long-running sparks and I'm spinning off more than the number of cores I have, even with depth=1. With depth=2 I did not get a run that could finish executing.
2. All the work in shards other than the "correct" one is wasted - we know they are never going to find the best solution. Unlike in the successful approach above, good explorations are not prioritized when the problem is divided at the top level. A* generally works by prioritizing exploration of good states rather than treating every state equally.
3. Execution is bottlenecked by the slowest spark, since every spark needs to finish before comparing solutions. Figure x below shows this mechanism happening (i.e. core 3 is first to finish, but core 0 needs to be waited on).



Repo

https://github.com/adelebai/a-star-tsp-haskell

Resources used:

- https://www.redblobgames.com/pathfinding/a-star/implementation.html to understand A* search in general
- https://www.public.asu.edu/~huanliu/AI04S/project1.htm to understand how to apply A* to TSP, and learning about MST heuristic
- https://en.wikipedia.org/wiki/Kruskal%27s_algorithm for reminder about how to implement Kruskal's algorithm

Code

Parallel Implementation (ParQueueProcessing.hs)

```haskell
tspSearch :: CityGraph -> Route
tspSearch cityGraph = Route (reverse (path bestNode))
  where
    cities = getCities cityGraph
    startCity = head cities  -- arbitrary
    goalState = Set.fromList cities
    initialNode = Node startCity [startCity] 0 0
    frontier = PQ.singleton initialNode
    visitedStates = HashMap.empty
    -- this is the max number of states to process in parallel, per iteration.
    k = 600
    -- maximum depth to explore for each node
    maxDepth = 1
    bestNode = parAstarSearch frontier visitedStates cityGraph goalState startCity k maxDepth


parAstarSearch :: PQ.MinQueue Node -> HashMap.HashMap (City, Set.Set City) Int -> CityGraph -> Set.Set City -> City -> Int -> Int -> Node
parAstarSearch frontier visitedStates cityGraph goalState startCity k maxDepth
  | PQ.null frontier = Node startCity [] 9999999999 9999999999 -- return a clearly bad solution instead of erroring.
  -- If the current front of the queue is the goal, we can exit.
  | isGoal currentNode goalState startCity = currentNode
  | otherwise = parAstarSearch updatedFrontier newVisitedStates cityGraph goalState startCity k maxDepth
  where
    -- only peek the currentNode for the purpose of checking exit condition.
    currentNode = fromJust (PQ.getMin frontier)
    -- take k items from the queue
    (nodeBatch, reducedFrontier)  = PQ.splitAt k frontier

    -- Filter out nodes that have been visited with a lower cost, or are goal states
    filterConditions = fmap (\n -> (n, isGoal n goalState startCity || not (hasVisitedBefore n visitedStates))) nodeBatch `using` parListChunk 10 rdeepseq
    filteredNodeBatch = [fst n | n <- filter snd filterConditions]

    expansions = fmap (\node -> depthLimitedExpansion node maxDepth cityGraph goalState) filteredNodeBatch `using` parListChunk 10 rdeepseq

    allNewNodes = PQ.unions expansions
    newVisitedStates = foldr (\n -> HashMap.insert (city n, Set.fromList (path n)) (gCost n)) visitedStates filteredNodeBatch
    updatedFrontier = PQ.union reducedFrontier allNewNodes

-- We consider a state visited if it's in the HashMap AND has a lower or equal cost than the current node.
hasVisitedBefore :: Node -> HashMap.HashMap (City, Set.Set City) Int -> Bool
hasVisitedBefore n visitedStates
  | HashMap.null visitedStates = False
  | otherwise = case HashMap.lookup element visitedStates of
                  Just cost -> gCost n >= cost
                  Nothing -> False
  where
    element = (city n, Set.fromList (path n))

-- depthLimitedExpansion should only generate successors, NOT explore them.
depthLimitedExpansion :: Node -> Int -> CityGraph -> Set.Set City -> PQ.MinQueue Node
depthLimitedExpansion startNode maxDepth cityGraph goalState = explore [startNode] 0
  where
    explore :: [Node] -> Int -> PQ.MinQueue Node
    explore [] _ = PQ.empty
    explore n_list k
      | k >= maxDepth = PQ.fromList n_list
      | otherwise =
          let goals = filter (\n -> isGoal n goalState 0) n_list
              successors = concat(fmap (\n -> expandNode n cityGraph goalState 0) n_list)
          in explore (successors ++ goals) (k+1)
```

Serial Implementation (Serial.hs)

```haskell
-- This is the serial implementation of A*.
tspSearch :: CityGraph -> Route
tspSearch cityGraph = Route (reverse (path bestNode))
  where
    cities = Serial.getCities cityGraph
    startCity = head cities  -- arbitrary
    goalState = Set.fromList cities
    initialNode = Node startCity [startCity] 0 (heuristic startCity [startCity] goalState cityGraph)
    frontier = PQ.singleton initialNode
    visitedStates = HashSet.empty
    bestNode = astarSearch frontier visitedStates cityGraph goalState startCity


-- Get all unique cities in the graph, which is just the keys of the graph.
getCities :: CityGraph -> [City]
getCities (CityGraph graph) = Map.keys graph

astarSearch :: PQ.MinQueue Node -> HashSet.HashSet (City, Set.Set City) -> CityGraph -> Set.Set City -> City -> Node
astarSearch frontier visitedStates cityGraph goalState startCity
  | PQ.null frontier = error "No solution found"
  | isGoal currentNode goalState startCity = currentNode
  | (currentCity, currentVisited) `HashSet.member` visitedStates = astarSearch restQueue visitedStates cityGraph goalState startCity
  -- | (currentCity, currentVisited) `Set.member` visitedStates' = astarSearch restQueue visitedStates' cityGraph goalState startCity
  | otherwise = astarSearch newFrontier newVisitedStates cityGraph goalState startCity
  where
    (currentNode, restQueue) = fromJust (PQ.minView frontier)
    currentCity = city currentNode
    currentVisited = Set.fromList (path currentNode)
    newVisitedStates = HashSet.insert (currentCity, currentVisited) visitedStates
    successors = expandNode currentNode cityGraph goalState startCity
    newFrontier = foldr PQ.insert restQueue successors
```

A-Star Functions (AStarLib.hs)

```haskell
isGoal :: Node -> Set.Set City -> City -> Bool
isGoal node goalState startCity =
    Set.fromList (path node) == goalState && city node == startCity && length (path node) > 1

getCities :: CityGraph -> [City]
getCities (CityGraph graph) = Map.keys graph

expandNode :: Node -> CityGraph -> Set.Set City -> City -> [Node]
expandNode node cityGraph goalState startCity = successors
  where
    (CityGraph graph) = cityGraph
    currentCity = city node
    nodePath = path node
    visitedCities = Set.fromList nodePath
    possibleEdges = Map.findWithDefault [] currentCity graph
    unvisitedEdges = filter (\(Edge dest _) -> dest `Set.notMember` visitedCities) possibleEdges
    successors = [createNode node edge goalState cityGraph | edge <- unvisitedEdges] ++ returnToStartNode
    returnToStartNode = if Set.size visitedCities == Set.size goalState && currentCity /= startCity
                        then createReturnNode node cityGraph startCity
                        else []
```

```haskell
-- Create a node that returns to the start city
createReturnNode :: Node -> CityGraph -> City -> [Node]
createReturnNode node cityGraph startCity =
    case edgeToStart of
        (Edge _ dist):_ -> [Node {
                                city = startCity,
                                path = startCity : path node,
                                gCost = gCost node + dist,
                                fCost = gCost node + dist
                            }]
        _ -> []
    where
        (CityGraph graph) = cityGraph
        currentCity = city node
        possibleEdges = Map.findWithDefault [] currentCity graph
        edgeToStart = filter (\(Edge dest _) -> dest == startCity) possibleEdges
```

Heuristic (AStarLib.hs)

```haskell
-- Heuristic function using MST over unvisited cities
heuristic :: City -> [City] -> Set.Set City -> CityGraph -> Distance
-- heuristic _ _ _ _ = 0
heuristic currentCity currentPath goalState cityGraph = mstCost relevantCities cityGraph
  where
    visitedCities = Set.fromList currentPath
    unvisitedCities = Set.difference goalState visitedCities
    relevantCities = Set.insert currentCity unvisitedCities

mstCost :: Set.Set City -> CityGraph -> Distance
mstCost cities (CityGraph graph) = totalCost
  where
    (totalCost, _) = kruskal sortedEdges initialParentMap 0
    initialParentMap = Map.fromList [(city', city') | city' <- Set.toList cities]
    sortedEdges = List.sortBy (\(_, _, d1) (_, _, d2) -> compare d1 d2) edges
    edges = getEdgesBetween cities graph

```

```haskell
type Edge3 = (City, City, Distance)

-- Get all edges between the given cities
getEdgesBetween :: Set.Set City -> Map.Map City [Edge] -> [Edge3]
getEdgesBetween cities graph = edges
  where
    edges = concatMap collectEdges (Set.toList cities)
    collectEdges u = [(u, v, d) | Edge v d <- Map.findWithDefault [] u graph,
                                  v `Set.member` cities,
                                  u <= v]  -- Avoid duplicates
--
kruskal :: [Edge3] -> Map.Map City City -> Distance -> (Distance, Map.Map City City)
kruskal [] parentMap accCost = (accCost, parentMap)
kruskal ((u,v,d):es) parentMap accCost =
    let rootU = find u parentMap
        rootV = find v parentMap
    in if rootU /= rootV
       then let updatedParentMap = union rootU rootV parentMap
                newAccCost = accCost + d
            in kruskal es updatedParentMap newAccCost
       else kruskal es parentMap accCost

-- Find in "Union-Find"
find :: City -> Map.Map City City -> City
find city' parentMap =
    let parent = Map.findWithDefault city' city' parentMap
    in if city' == parent
       then city'
       else find parent parentMap

-- Union in "Union-Find"
union :: City -> City -> Map.Map City City -> Map.Map City City
union = Map.insert
```

Data structures (structures.hs)

```haskell
type City = Int
type Distance = Int

-- Edge consist of its index (Int), and a distance (Int)
data Edge = Edge Int Distance deriving (Eq, Show, Read)

-- Data structure representing graph of cities. It's an Map of Lists
-- Note, we expect small number of cities & edges, so O(logk) access should be negligible
-- Using Lists at the 2nd level for iterative traversal of neighbours.
data CityGraph = CityGraph (Map City [Edge]) deriving (Eq, Show, Read)

-- Given a list of edges (node, node, distance), return a CityGraph.
getCityGraphFromEdges :: [(City, City, Distance)] -> CityGraph
getCityGraphFromEdges [] = CityGraph empty
getCityGraphFromEdges edgeList = CityGraph edgeMap
    where edgeMap = fromListWith (++) (fmap (\(n1,n2,d) -> (n1, [(Edge n2 d)])) edgeList)

-- Route is just a list of Nodes (Indicies)
data Route = Route [City] deriving (Eq, Ord, Show, Read)

getRouteFromString :: String -> Route
getRouteFromString [] = Route []
getRouteFromString s = Route (fmap (\x -> read x :: Int) (splitOn " " s))

getEdgesFromLines :: [String] -> [(Int, Int, Int)]
getEdgesFromLines [] = []
getEdgesFromLines l = let splitList = fmap (\x -> splitOn "," x) l
                      in fmap (\x -> (read (x !! 0) :: Int, read (x !! 1) :: Int, read (x !! 2) :: Int)) splitList

-- States in A* search
data Node = Node {
    city :: City,
    path :: [City],
    gCost :: Distance,    -- Cost so far
    fCost :: Distance     -- Estimated total cost (gCost + heuristic)
} deriving (Show)

-- this part needed to be compatible with deepseq.
instance NFData Node where
    rnf (Node c p g f) = rnf c `seq` rnf p `seq` rnf g `seq` rnf f

-- Equality and ordering for priority queue
instance Eq Node where
    n1 == n2 = fCost n1 == fCost n2
instance Ord Node where
    compare = compare `on` fCost
```