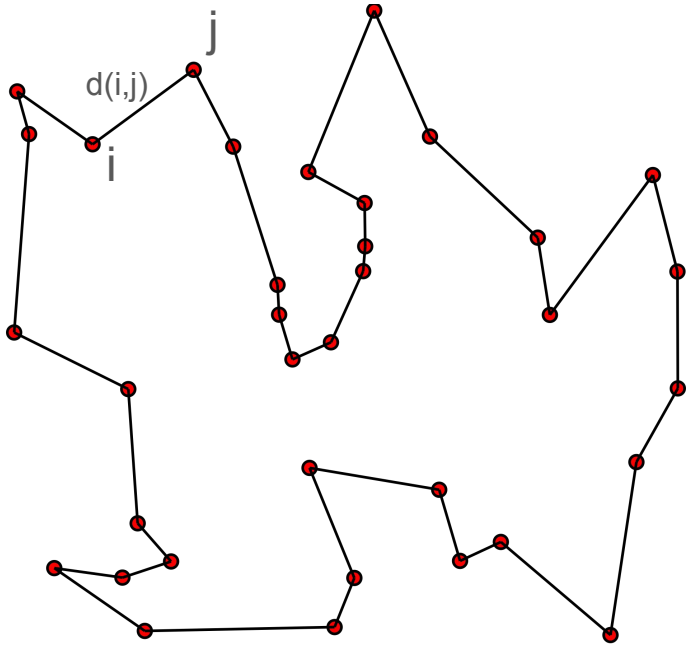
A* Search for TSP

Adele Bai (ayb2121)
Vincent Mutolo (vm2724)

Traveling Salesman

- Given cities i and distances between them d(i,j)
  - Note: we do not assume a metric space
- Goal: find shortest tour through all the cities. I.e. find a permutation of [0, n) representing the order of cities visited that minimizes the total distance travelled
- TSP is NP-hard
  - No metric, so can't even use 2-approximation
  - Naive solution requires O(n!) time because n! permutations
  - Dynamic programming in general requires O(n^2 x 2^n) time



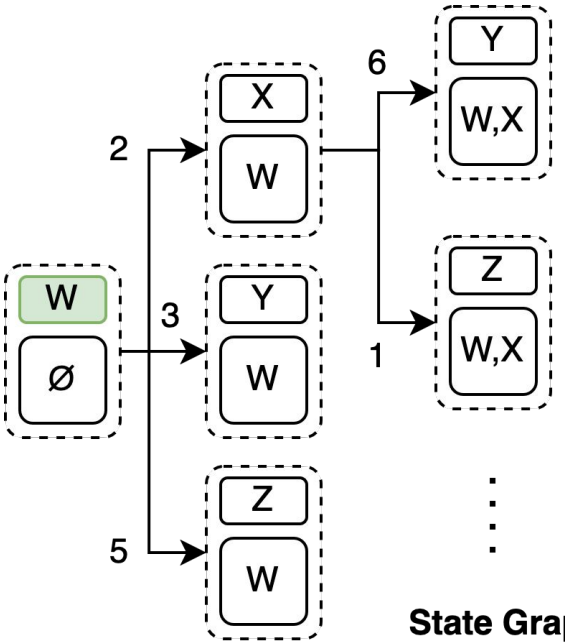
https://en.wikipedia.org/wiki/Travelling_salesman_problem

Serial Solution: A*

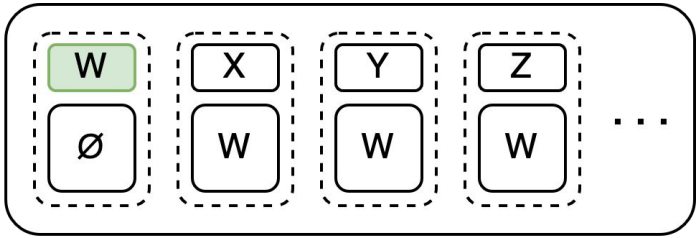
A* overview

A* state

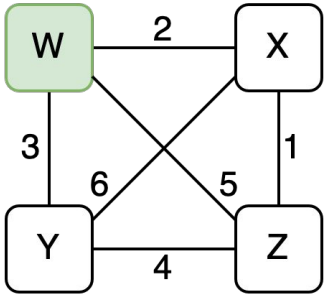
```
data Node = Node {
    city :: City,
    path :: [City],
    gCost :: Distance,
    fCost :: Distance
} deriving (Show)
```



State Graph



Priority Queue of States



City Graph

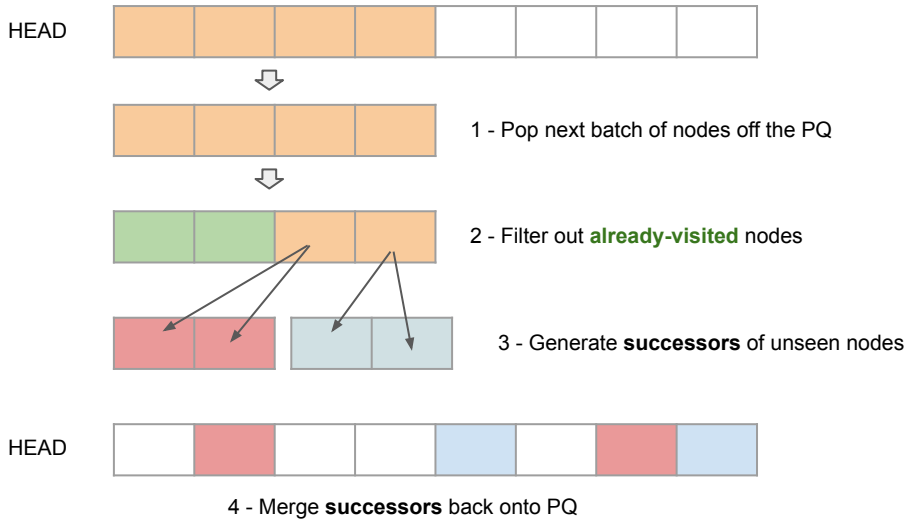
Parallelization approach

General approach - process more items from the queue in parallel.

In the diagram, step **(2)** and **(3)** can be easily parallelized since they are operations on lists.

Used the Control.Parallel.Strategies library since it had convenient methods like:

- ParListChunk
- rdeepseq



Parallelization approach (challenges and solutions)

1. **Correctness** - the visited states check must now be a HashMap to store the best cost equivalent states.
2. Forcing deepseq evaluation of lambdas every iteration.
3. Control number of sparks (10k-20k) by tuning **batch size** and **list chunk** size
   a. Without heuristic 2400/200 split worked best
   b. With heuristic 600/10 split worked best
4. Improving parallel GC throughput by using -A32m flag (default is 4MB)

```
-- 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
```

```
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)
```

Before (4MB alloc)

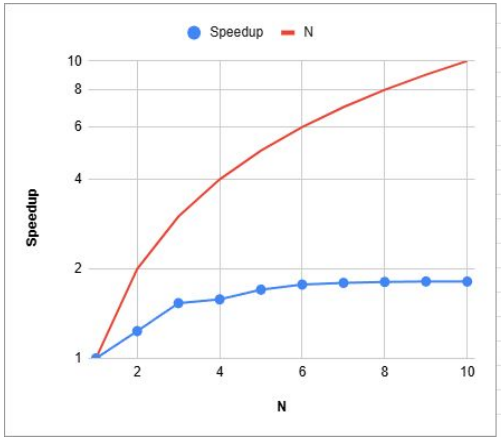

```
Unset
stack run 'data/17_cities_edges.csv' 'parattempt3' -- +RTS -N8 -s -A32m
Parallel GC work balance: 95.13% (serial 0%, perfect 100%)
  INIT    time    0.007s  (  0.028s elapsed)
  MUT     time   33.093s  ( 16.267s elapsed)
  GC      time   13.885s  (  2.162s elapsed)
  EXIT    time    0.044s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

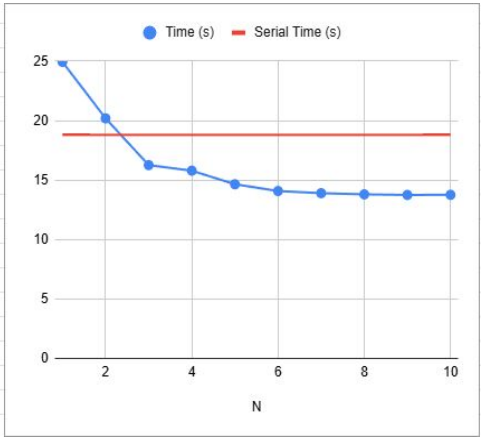
After (32MB alloc)

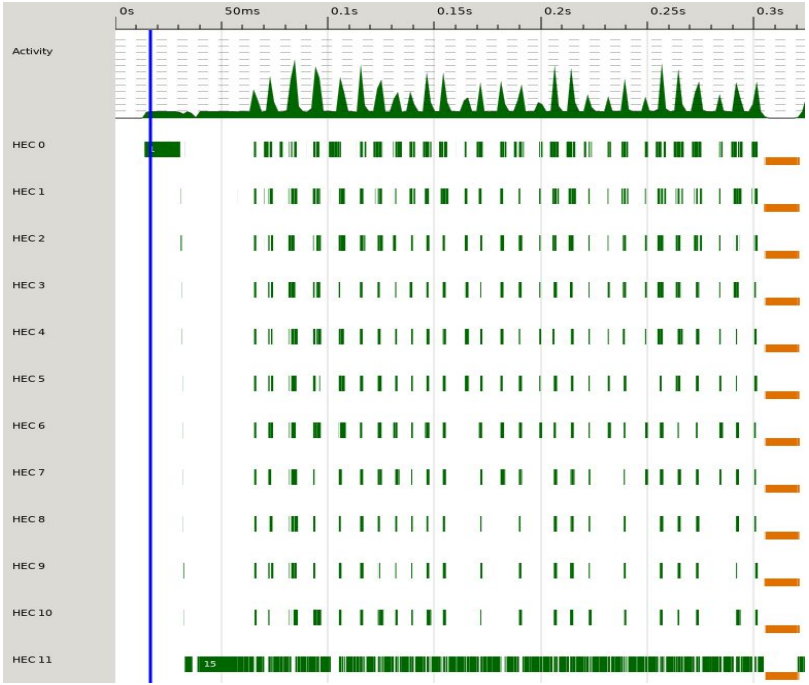
Initial parallelization results

Verdict: It's fine, could be better.

- Beats serial implementation by 25% at N=3
- Struggles to hit 2x scaling
- Bottlenecked by HashMap lookup (visited states check)
- Sparse core utilization on threadscope





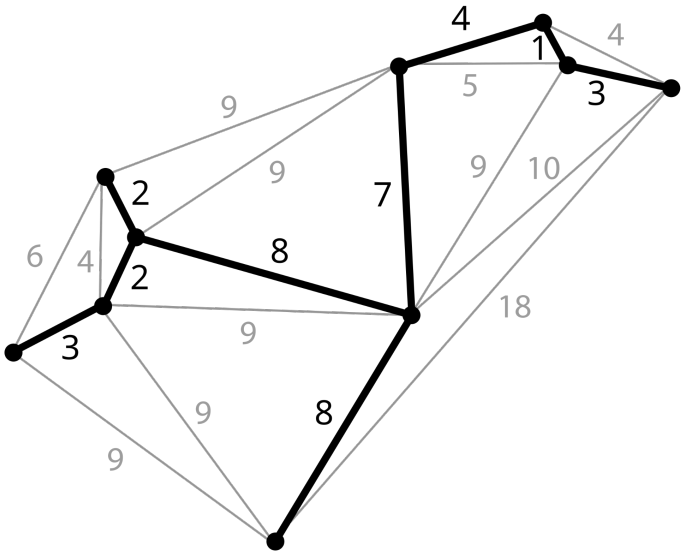


Cost Centers Profiling summary

```
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)
```

Improvement idea: MST heuristic

- MST must be a lower bound for remaining tour cost because it visits every node once, but isn't restricted in in/out degree
- If we use MST heuristic, every exploration will begin by calculating MST of the remaining nodes
    - This is relatively expensive, so it will shift bottleneck to MST
    - **Good because different MST calculations are done in parallel**



https://en.wikipedia.org/wiki/Minimum_spanning_tree

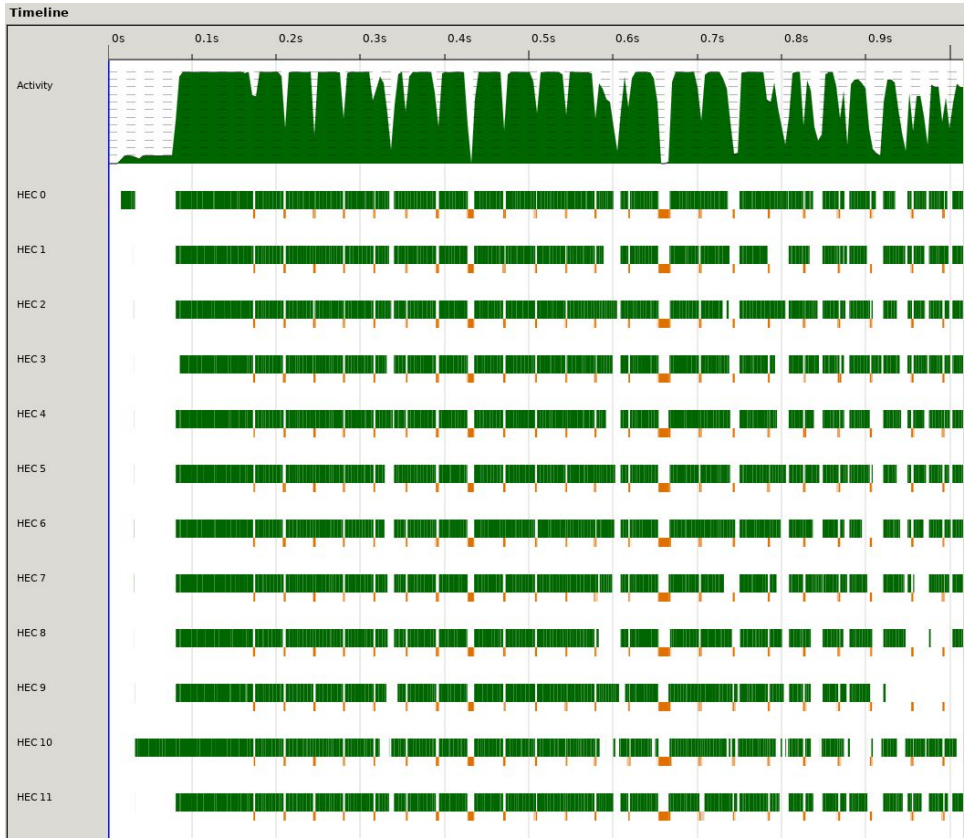
Impact of adding Heuristic?

More work to generate successors, BUT:

- Better PQ ordering (less nodes explored overall)
- HashMap.lookup no longer the bottleneck
- Overall 2-4x execution speedup.
- Better scaling and core utilization (see threadscope chart)

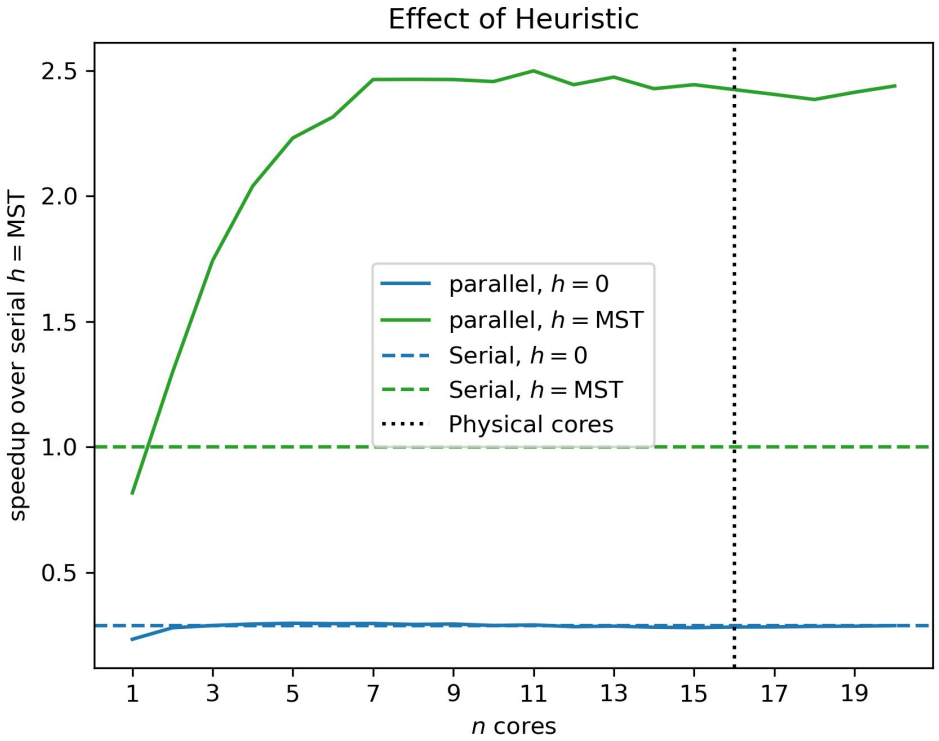
Cost Centers Profiling summary

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

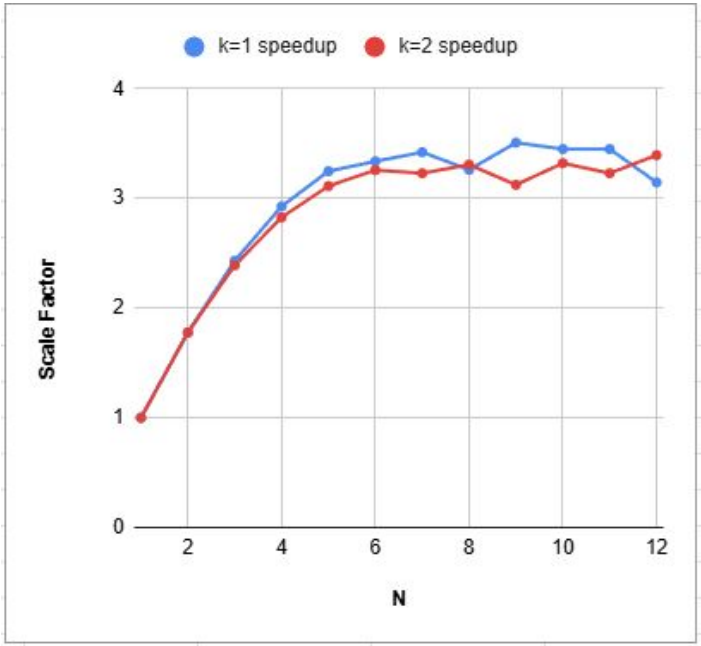


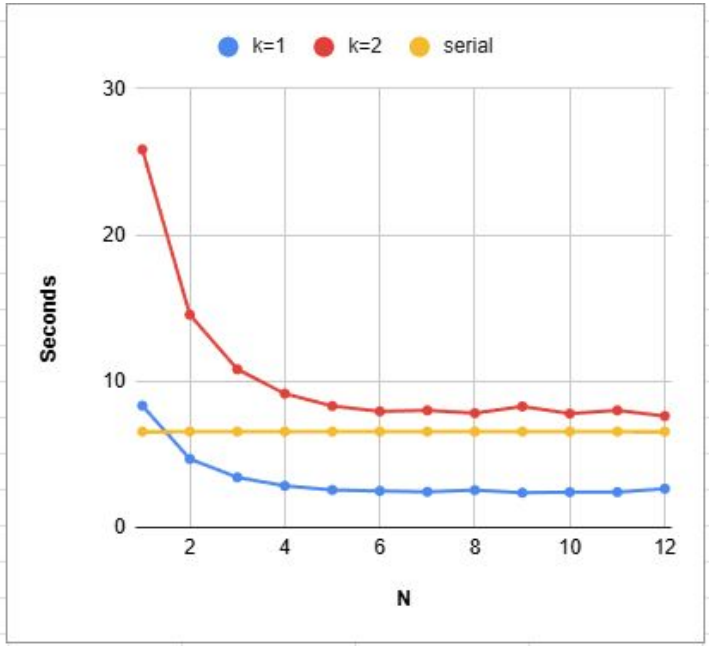
Improvement: MST heuristic

- 4x speedup for serial
- Much better scaling
  - 0.5N for h=MST, N=1..5
- “Saturation scale” is better
  - 2.5x with MST
  - 1.2x w/o MST



Tried: Generating successors of depth k + heuristic added



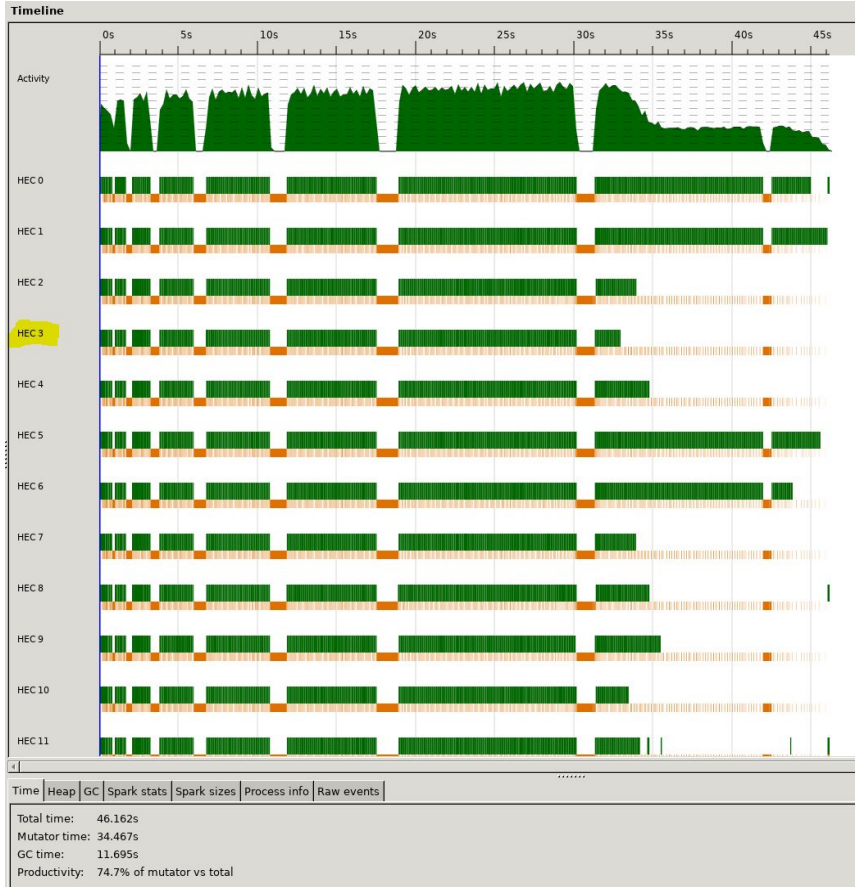


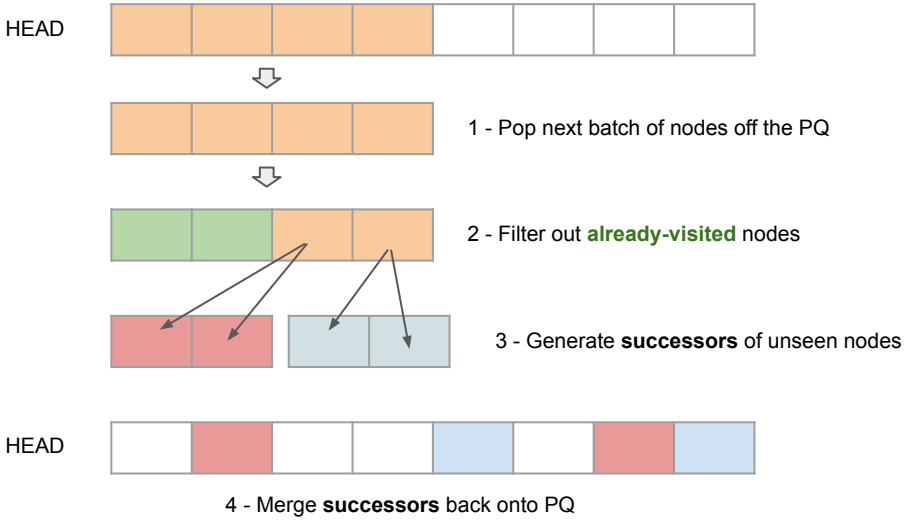
With heuristic, we're able to hit ~3.5x scaling.
With more depth, we get similar scaling but worse performance than single-depth exploration.

Other approaches that didn’t work - Naive Sharding

Shard the problem at the top-level, depth=1:

- Small number of long-running sparks generated
- Scaled well, but much slower than serial execution
- **Key issue:** all sparks are ‘equal’ so most of them are wasted.




HEAD
1 - Pop next batch of nodes off the PQ
2 - Filter out already-visited nodes
3 - Generate successors of unseen nodes
HEAD
4 - Merge successors back onto PQ

```haskell
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