CSSE2002/7023

Programming in the large

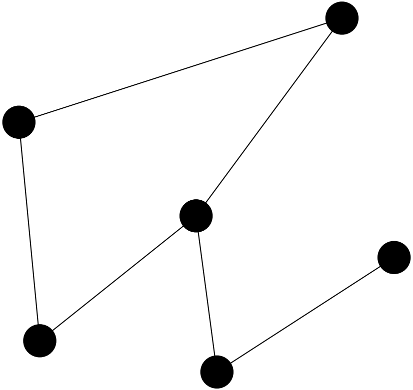
Week 8.1: Depth-first search and breadth-first search

In this hour ...

- Graphs
- Searching
- Depth-first search
- Breadth-first search

Graphs

A graph consists of a set of nodes / vertices and the connections between those nodes.



Graphs appear in programs whenever connections between objects need to be represented. (e.g., Tile in Assignment 1).

Graphs in Java

What does a graph look like in Java?

`City.java`

Searching graphs

How do we iterate through all the cities?
Each City only has references to its neighbours.

```
Brisbane      Sydney        Melbourne
- Sydney      - Melbourne   - Sydney
- Darwin      - Adelaide    - Adelaide

Adelaide      Perth         Darwin
- Melbourne   - Adelaide    - Brisbane
- Perth       - Darwin      - Perth
```

We can access a city through:

```
brisbane.getNeighbours().get(0).getCity().
getNeighbours().get(0).getCity() ...
```

Searching graphs

We normally need ...

- ... to know what we have already visited.
- ... to know the next few nodes we should be visiting.

Depth-first searching

Before we start:

- Create a Stack (*nodesToVisit*) containing a starting node.
- Create a Set (or Map) (*alreadyVisited*) that is empty.

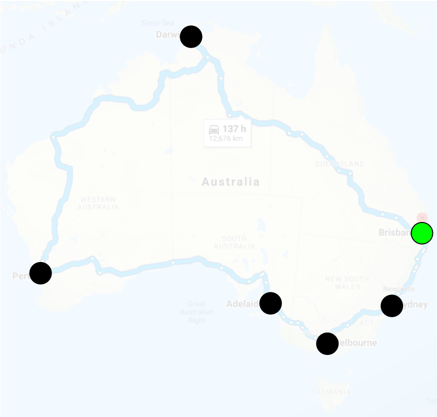
Then repeat the following until *nodesToVisit* is empty:

1. pop the next node from *nodesToVisit*.

   If the node is not in *alreadyVisited*:

   2. add the node to *alreadyVisited*.
   3. process the node.
   4. add each of the node's neighbours to *nodesToVisit*

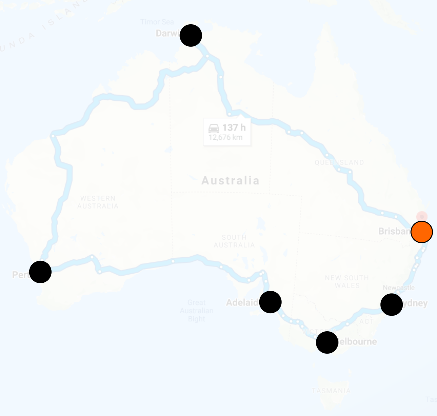
Depth-first searching



nodesToVisit:
Brisbane

alreadyVisited:

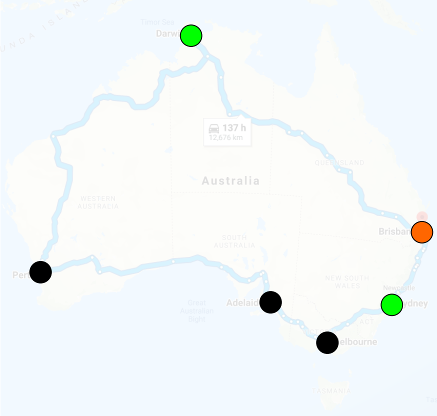
Depth-first searching



nodesToVisit:

alreadyVisited:

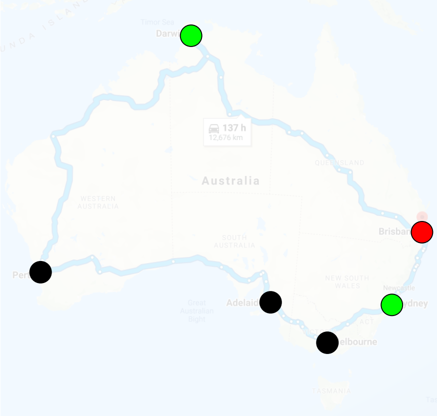
Depth-first searching



nodesToVisit:
Darwin
Sydney

alreadyVisited:

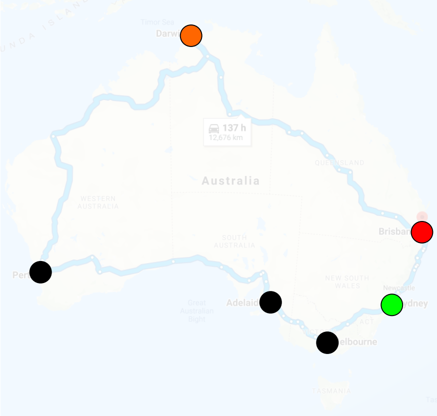
Depth-first searching



nodesToVisit:
Darwin
Sydney

alreadyVisited:
Brisbane

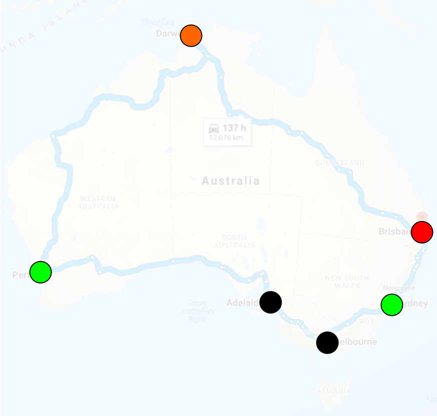
Depth-first searching



nodesToVisit:
Sydney

alreadyVisited:
Brisbane

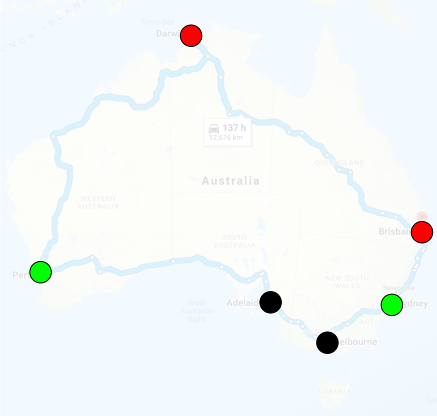
Depth-first searching



nodesToVisit:
Perth
Sydney

alreadyVisited:
Brisbane

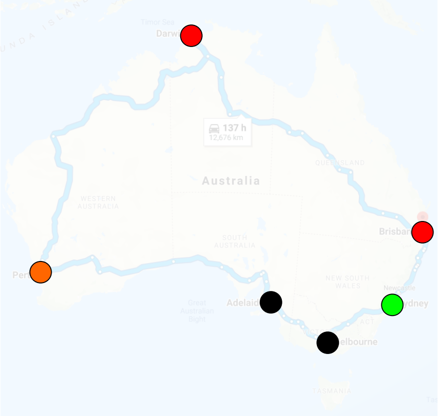
Depth-first searching



nodesToVisit:
Perth
Sydney

alreadyVisited:
Brisbane
Darwin

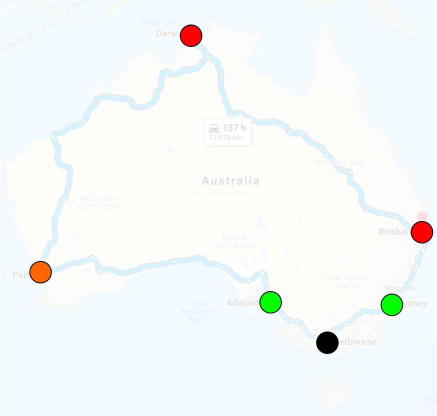
Depth-first searching



nodesToVisit:
Sydney

alreadyVisited:
Brisbane
Darwin

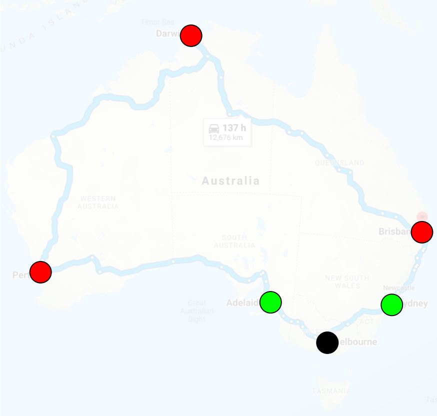
Depth-first searching



nodesToVisit:
Adelaide
Sydney

alreadyVisited:
Brisbane
Darwin

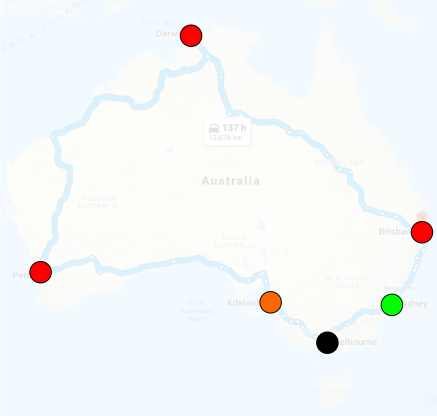
Depth-first searching



nodesToVisit:
Adelaide
Sydney

alreadyVisited:
Brisbane
Darwin
Perth

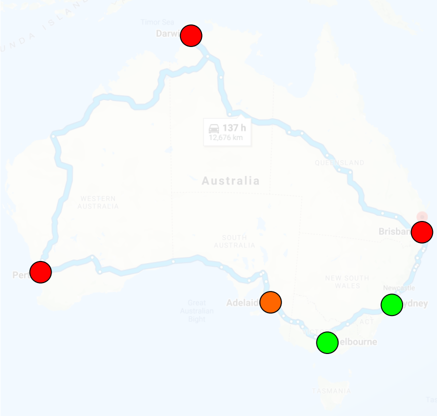
Depth-first searching



nodesToVisit:
Sydney

alreadyVisited:
Brisbane
Darwin
Perth

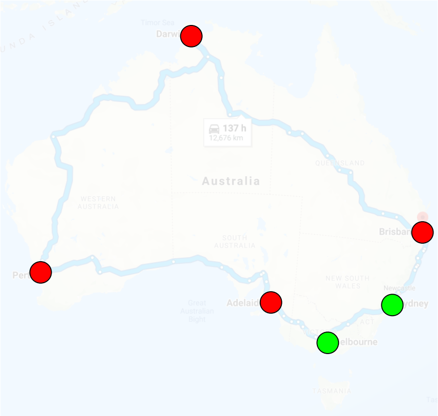
Depth-first searching



nodesToVisit:
Melbourne
Sydney

alreadyVisited:
Brisbane
Darwin
Perth

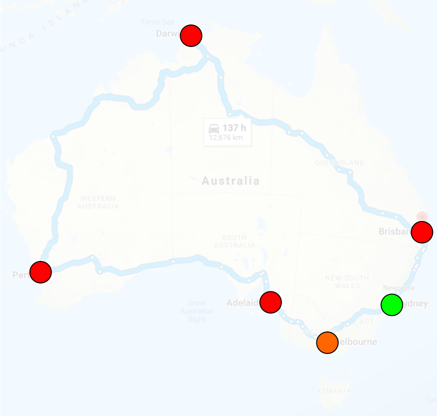
Depth-first searching



nodesToVisit:
Melbourne
Sydney

alreadyVisited:
Brisbane
Darwin
Perth
Adelaide

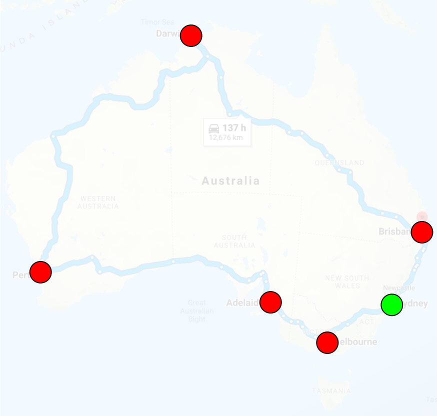
Depth-first searching



nodesToVisit:
Sydney

alreadyVisited:
Brisbane
Darwin
Perth
Adelaide

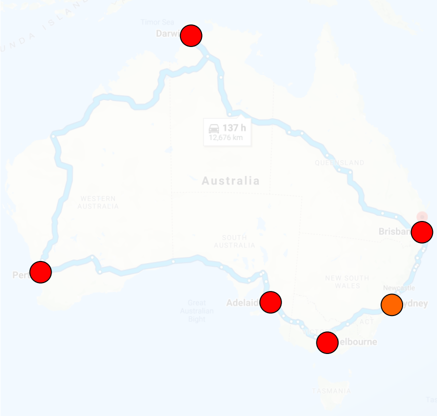
Depth-first searching



nodesToVisit:
Sydney

alreadyVisited:
Brisbane
Darwin
Perth
Adelaide
Melbourne

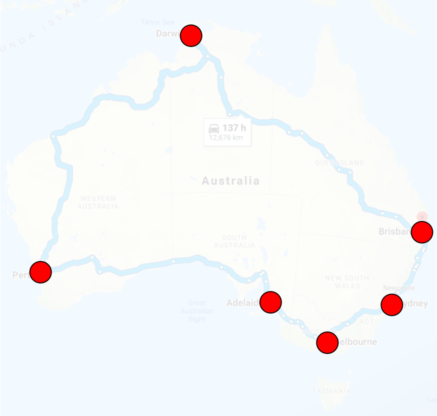
Depth-first searching



nodesToVisit:

alreadyVisited:
Brisbane
Darwin
Perth
Adelaide
Melbourne

Depth-first searching



nodesToVisit:

alreadyVisited:
Brisbane
Darwin
Perth
Adelaide
Melbourne
Sydney

Depth-first searching

`SearchCities.java`

Breadth-first searching

What if we want to instead process nodes in the same order as their links to the starting node?
We might want to ...

- ... look at closer options before we look at those that are further away.
- ... find the shortest number of connections between two nodes.

Depth-first searching

Before we start:

- Create a Stack (*nodesToVisit*) containing a starting node.
- Create a Set (or Map) (*alreadyVisited*) that is empty.

Then repeat the following until *nodesToVisit* is empty:

1. pop the next node from *nodesToVisit*.

   If the node is not in *alreadyVisited*:

   2. add the node to *alreadyVisited*.
   3. process the node.
   4. add each of the node's neighbours to *nodesToVisit*

Breadth-first searching

Before we start:

- Create a **Queue** (*nodesToVisit*) containing a starting node.
- Create a Set (or Map) (*alreadyVisited*) that is empty.

Then repeat the following until *nodesToVisit* is empty:

1. pop the next node from *nodesToVisit*.

   If the node is not in *alreadyVisited*:

   2. add the node to *alreadyVisited*.
   3. process the node.
   4. add each of the node's neighbours to *nodesToVisit*

Queue

- F.I.F.O.
- An interface in Java collections `java.util.Queue`
  (Note: for some reason `java.util.Stack` is a class)
- Has a number of implementations - two common ones are `java.util.LinkedList` and `java.util.ArrayDeque`.
- Follows the metaphor of a queue of people. People enter at one end and leave at the other, and the first to arrive is the first to leave.

Breadth-first searching

`SearchCities2.java`

Finding the shortest distance

- There are many other graph searching algorithms that vary the order of visiting nodes (Greedy search, A*, Dijkstra's algorithm, Iterative deepening, etc.)
- Dijkstra's algorithm visits the closest points first (by distance, not by links).
- When a node is visited for the first time, the distance taken is guaranteed to be the shortest.
- Instead of using a Queue or a Stack, we have to sort `nodesToVisit` each iteration.

Sorting lists in Java

- `List.sort()` or `Collections.sort(List)`
- `list.sort(null);` - all elements must implement Comparable interface
- `list.sort(comparator);` - the list is sorted using a Comparator interface.

  The Comparator interface requires that the method `Comparator.compare(T o1, T o2)` is overridden.

Shortest distance

`CalcCityDistances.java`