# How to find your way with algorithms.

Jonathan Tsai

University of Hong Kong

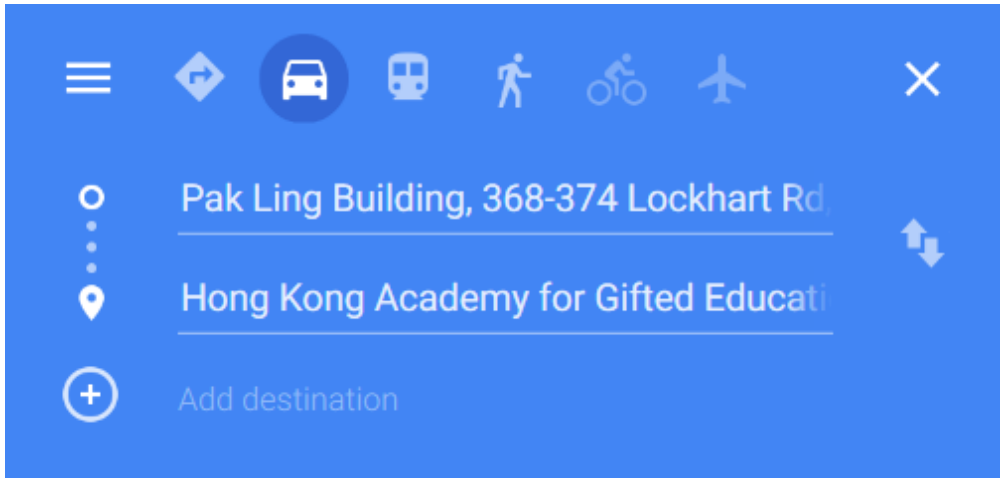
jtsai@giftofmaths.com

16 August, 2019

# How to get to the HKAGE

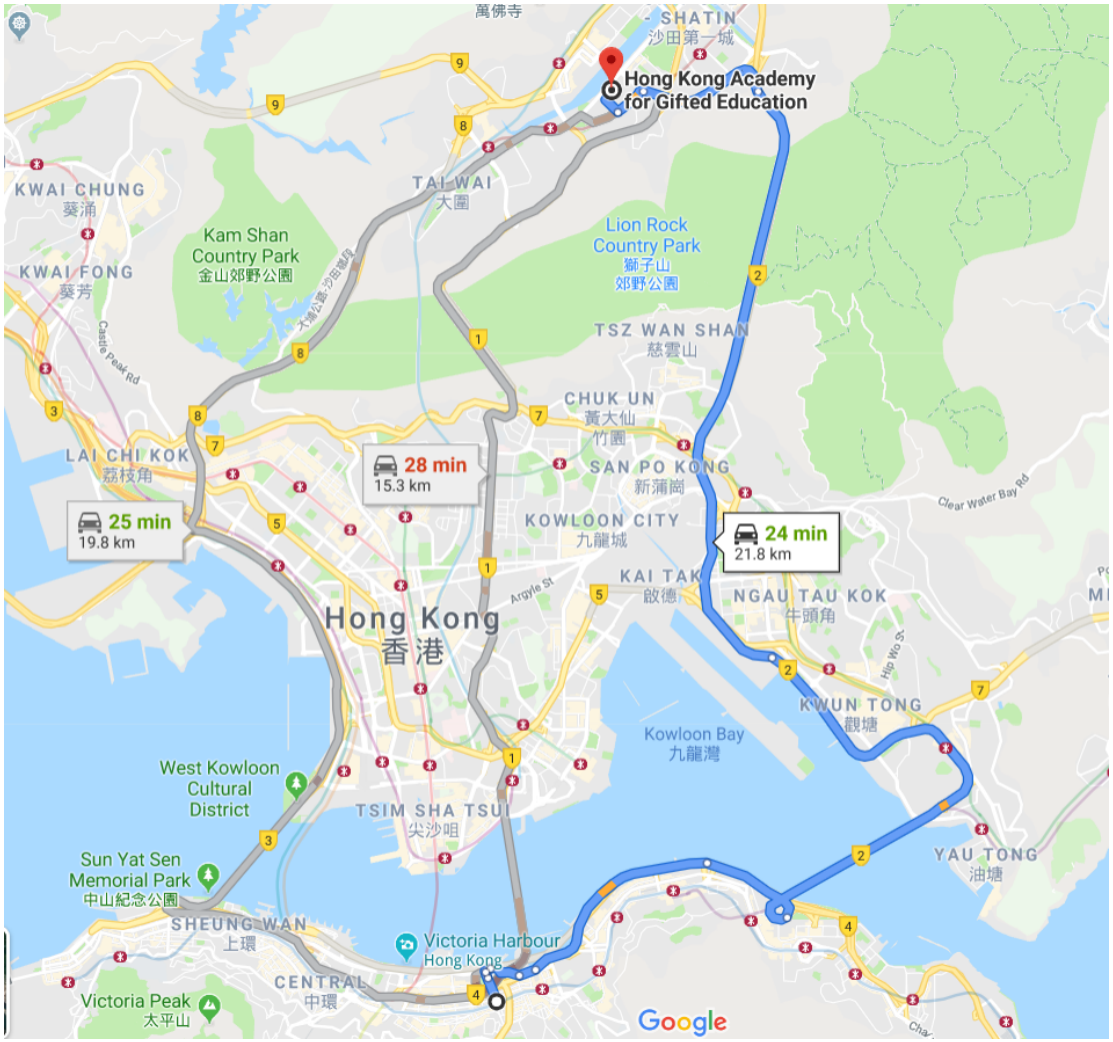
I wanted to find the quickest way to get to the Hong Kong Academy of Gifted Education by car.

I do what everyone does and open Google Maps. I type in my current location and the target destination.



# Google Maps

Google Maps gives me the quickest route to the HKAGE. I can also change the settings so that it gives the shortest route by distance.



# Main question

Today, we will address the following question

**Question**

How does Google Maps do this?

Clearly, computers don't think (yet!). So it must be programmed to be able to do this.
What we mean here is that we must program the computer to follow a specific set of instructions.

These instructions will find us what we want i.e. the shortest/quickest route from my office to the HKAGE. We can rephrase our question better.

**Question**

What is the set of instructions that allow us to find the shortest/quickest route between two places.

# Algorithms

These set of instructions are called an **algorithm**.

The word algorithm is named after the Persian mathematician Al-Khwarizmi (c. 780 - c. 850). He wrote a book which detailed specific sets of instructions for solving equations.

We use algorithms all the time in our everyday life:

- When we cook.
- When we get dressed.
- When we take the MTR.

# Shortest path algorithms

We return to the example of Google Maps. What are the set of instructions?

## A naïve algorithm

Steps:

1. List out all possible routes from the starting point to the ending point.
2. Add up the total distances of each route.
3. Find the smallest total distance. The route corresponding to this total distance is the shortest path.

Is this a good algorithm?

Why or why not?

# Problems with the naïve algorithm

Firstly, we need to give specific instructions on how to do step 1.
Remember: a computer cannot think!

## Main Problem

The list of all possible routes would be really long. Assuming that each intersection their is 3 choices: turn left, turn right, and go straight ahead. If there is only one intersection, there would be 3 possible paths. If there are 2 intersections, there are 9 possible paths. If there are 20 intersections, the number of paths would be

$$3^{20} = 3486784401$$

It would take a long time to create such a long list.

However, there is one very important good thing about this algorithm.

# Considerations when creating algorithms

The good thing about the naïve algorithm is that it always works and is guaranteed to give you the shortest route.

The bad thing is that it would take a very (very) long time to run.

## Main consideration

- We want to design an algorithm in a way that can be understood by a computer.
- We want to design algorithms that are quick.
- We want an algorithm that works.

# Game

We are going to play a game.

## Rules

1. In the next slide, you will be given a list of words. You have to sort them in alphabetical order (A - Z).
2. Write your answers on a sheet of paper.
3. The final answer must be in the form of a list.
4. Please raise your hand when you have completed the task.
5. The first person to raise their hand and provide the correct answer wins.
6. There will be (very small) prize for the winner.

Sort the following list of 25 words into alphabetical order (A to Z).

| | | | | |
|---|---|---|---|---|
| rent | slim | car | bird | salt |
| wide | cash | ahoy | buy | sea |
| rear | skin | bet | vet | try |
| snip | heel | ring | hack | iron |
| nice | more | sing | dock | from |

# Sorting algorithms: Insertion sort

We now look at algorithms for sorting things in to order (numerical, alphabetical etc.)

Imagine you had a deck of playing cards and you were sorting into order. Suppose we consider just the hearts. Most people would do this by taking out cards and inserting them into the right place. This is essentially our first algorithm: **insertion sort**.

## Insertion sort

Steps:

1. Look at the first 2 items. Put them in the right order.
2. Look at the next item, insert this item in the correct position relative to the previous items.
3. Repeat with each subsequent item until you reach the end of the list.

# Sorting algorithms: Bubble sort

The insertion sort is intuitive. But we now consider other sorting algorithms.

The **bubble sort** interchanges pairs of items from left to right. This is not enough to sort a list. What we do is repeat this step until the list is sorted.

## Bubble sort

Steps:

1. Compare the 1st and 2nd items. Either leave them or swap them (to put in right order).
2. Now compare the 2nd and 3rd items. Either leave them or swap them.
3. Continue with 3rd and 4th item,... until we reach the end of the list.
4. Repeat steps 1 to 3 until you go through the whole list without any swaps.

# Sorting algorithms: Quicksort

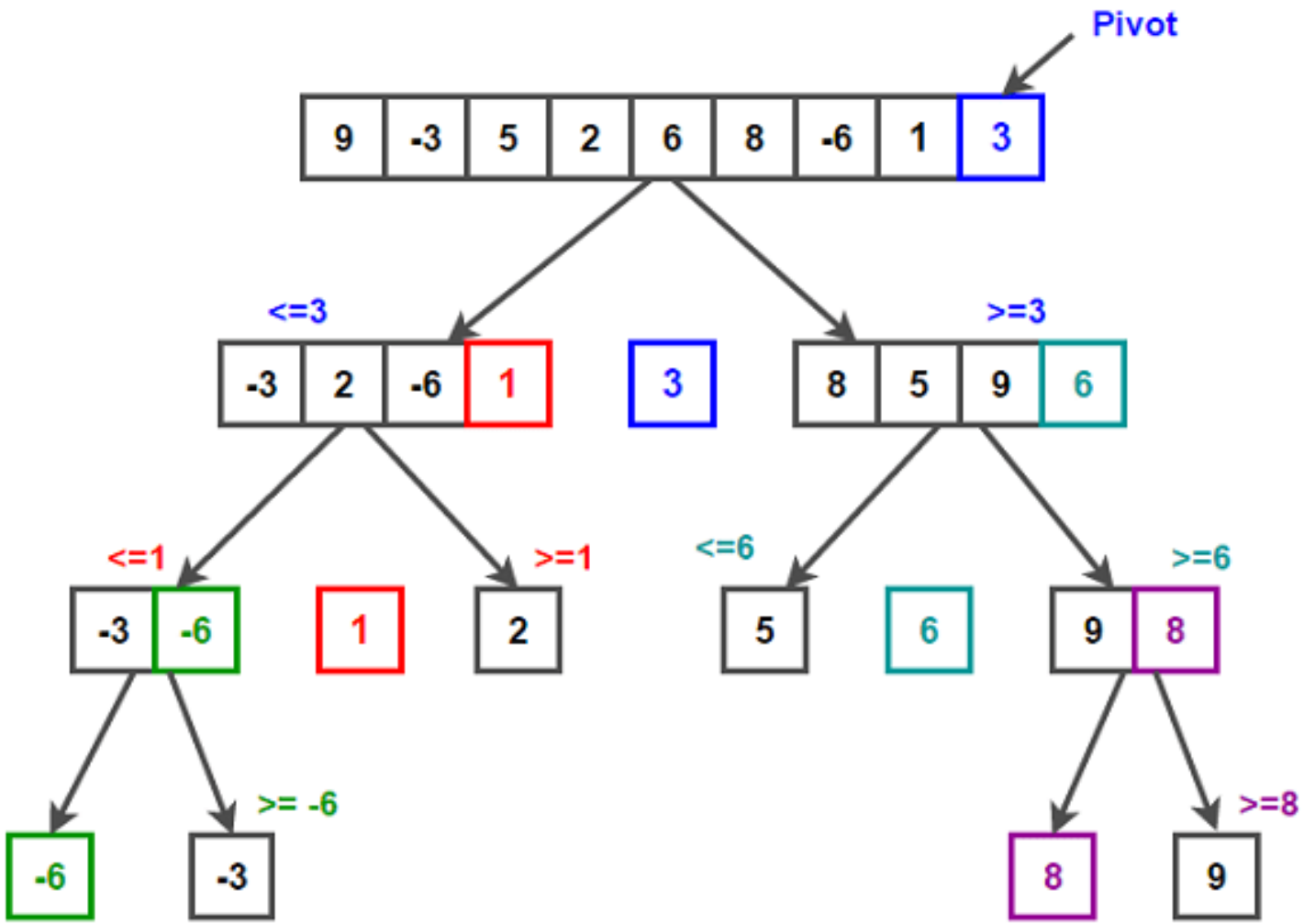
We look at one final example of a sorting algorithm: **quicksort**

## Quicksort

Steps:

1. Take an item $p$ in the list. $p$ is called the *pivot*
2. Divide the list into two sublists:
   - Items less than or equal to $p$ which we move to the left of $p$.
   - Items items more that $p$ which we move to the right of $p$.
3. Repeat steps 1 and 2 to each sublist to get further sublists.
4. Stop when every entry has been chosen as a pivot.

# Sorting algorithms: Quicksort



# Speed of an algorithm I

How do we compare the speed of an algorithm? Take sorting algorithms as an example.

Suppose that a "normal" list of $N$ items takes approximately a certain time $T_1$ to compute; and a "normal" list of $10 \times N$ items takes $T_2$ to compute. We consider the ratio $\frac{T_2}{T_1}$.
When $N$ is large, we would expect $\frac{T_2}{T_1}$ to be around some power of 10 i.e. $10^d$. It is this number $d$ that we use to compare the algorithms.

e.g. Using a particular algorithm: if a sorting a list of 100,000 items takes 20 seconds and sorting a list of 1,000,000 items takes 2000 seconds, then:

$$\frac{2000}{20} = 100 = 10^2 \Rightarrow d = 2$$

But what is meant by normal?

# Speed of an algorithm II

To calculate and compare speeds, we usually consider either: the **best case** speed, the **worst case** speed, the **average case** speed.

| Algorithm | Best Case | Average Case | Worst case |
|---|---|---|---|
| Insertion Sort | $d = 1$ | $d = 2$ | $d = 2$ |
| Bubble Sort | $d = 1$ | $d = 2$ | $d = 2$ |
| Quicksort | $1 < d < 1.1$ | $1 < d < 1.1$ | $d = 2$ |

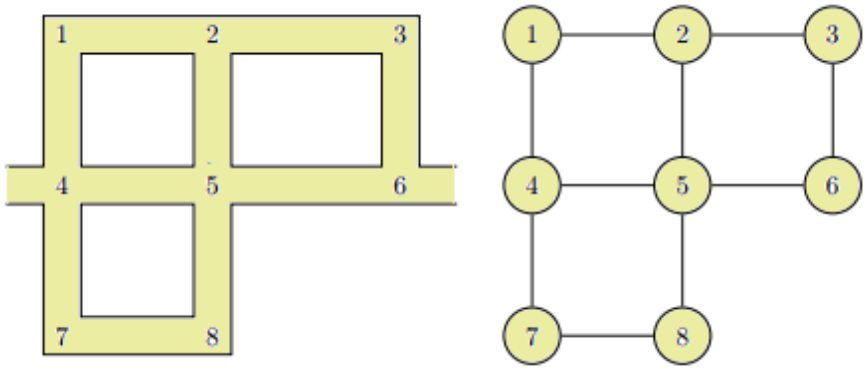
Remark:

- From this table, it may look like quicksort is the best. Except in the best case situation - this corresponds to lists already sorted.
- The speed is only applicable for large lists. For short lists, the insertion sort is faster than the other two.
- Quicksort has another advantage which is that it creates sublists. These can be split between multiple "cores".

# Graphs and Networks

We now return to shortest path algorithms. The first problem we encounter is how do we convert a map to something a computer can understand?

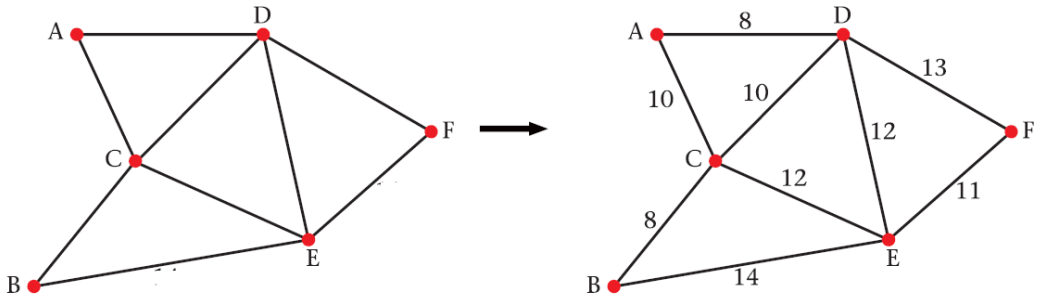
The trick is to convert the map to a graph (not the normal kind!). A **graph** is a collection of nodes joined by edges.




We are not interested in the position of the nodes. Only how they are joined to each other.

# Graphs and Networks

On each edge of the graph, we can add a number representing distance (or time) - this number represents the distance (or time) to cross the edge. This results in a **network**.




We can construct a *distance matrix* from the network:

| | A | B | C | D | E | F |
|---|---|---|---|---|---|---|
| A | - | - | 10 | 8 | - | - |
| B | - | - | 8 | - | 14 | - |
| C | 10 | 8 | - | 10 | 12 | - |
| D | 8 | - | 10 | - | 12 | 13 |
| E | - | 14 | 12 | 12 | - | 11 |
| F | - | - | - | 13 | 11 | - |

e.g. the entry in the "A" row and "C" column represents the edge of length 10 between vertices A and C. We store the information on the map as this distance matrix.

# Dijkstra's algorithm

An example of a shortest path algorithm is Dijkstra's algorithm from a starting node $S$ to an destination node $T$.

## Dijkstra's algorithm

Steps:

1. A node is either **Unvisited** and **Visited**. Every node is unvisited at the beginning. Assign a temp. distance value of 0 to the $S$ and $\infty$ to all other nodes.
2. The current node is $S$.
3. For the current node, consider all of its unvisited neighbours and calculate their temp. distances to the current node. Compare the newly calculated temp. distance to the current assigned value and assign the smaller one.
4. When we are done considering all of the unvisited neighbours of the current node, mark the current node as visited.
5. Select the unvisited node that is marked with the smallest temp. distance, set it as the new "current node".
6. Repeat steps 3 to 6 until the destination node has been marked visited.

In the interest of time, we will just see it in action on the animation.

# Dijkstra's algorithm

Dijkstra's algorithm is an algorithm on the network. It can be shown that it always gives us the shortest route.

We created the network from the map by creating a graph and adding numbers on the edges to represent distance. All other information from the real map is discarded.

Remember: computer's cannot think!

Everyone here knows that to get from Wan Chai to HKAGE, I do not need to go through Lantau Island. Dijkstra's algorithm doesn't take this into account - it would consider routes that would be ridiculous to any human. This fact slows down the algorithm.

# How can we improve it?

For a human, we can decide whether a potential path is ridiculous just by "guessing". This "guess" is called a **heuristic**.

By taking into account a heuristic in the algorithm, the computer can avoid considering all paths and just consider the ones that are likely to be correct (i.e. the "guesses").

A shortest path algorithm which takes a heuristic into account is called the A* algorithm.
It is a lot faster than Dijkstra's algorithm. However, for certain heuristics, the A* algorithm is not able to find the shortest path.

Does this matter?

# Summary

- Algorithms are set of instructions that can be followed in order to achieve a desired result e.g. find the shortest path, sort lists,...
- Algorithms are used throughout daily life - from simple one to complicated ones.
- Considerations when designing algorithms:
  - Speed
  - Accuracy
- There may be several algorithms that do the same thing - each may be suited for different tasks. There is often no "best" algorithm.

# The end

Thanks for listening!

Any questions?

jtsai@giftofmaths.com