Classifying Computational Problems

Murray Cole

Machines and Algorithms

In CS1, we have seen algorithms for various problems on two different types of machine:

- Finite State Machine (represented by FSM diagrams)
- “Conventional Computer” (represented by Java programs)

For each machine, certain problems seem harder than others, and some may be impossible to solve.

- searching a sorted array is “easier” than sorting the array
- checking for equal numbers of A’s and B’s in the input is impossible for a FSM

Problem Classes

Classifying problems by their difficulty, and noting how members of each class are related, can be useful

- stops us from trying to solve the impossible
- suggests that a problem might be so hard (but not impossible) that we should accept “nearly right” solutions

Sometimes knowing that a problem is very hard can be a positive thing - many cryptographic algorithms are only assumed secure because certain mathematical problems are very hard to solve.

Machine Models

In absolute terms, any real computer is just a Finite State Machine!

- fixed number of memory locations each storing a fixed size piece of data
- fixed number of transistors in the CPU etc
- so, a finite (but huge) number of possible states

For example, in the extreme, no real computer can check matching A and B counts in the input.

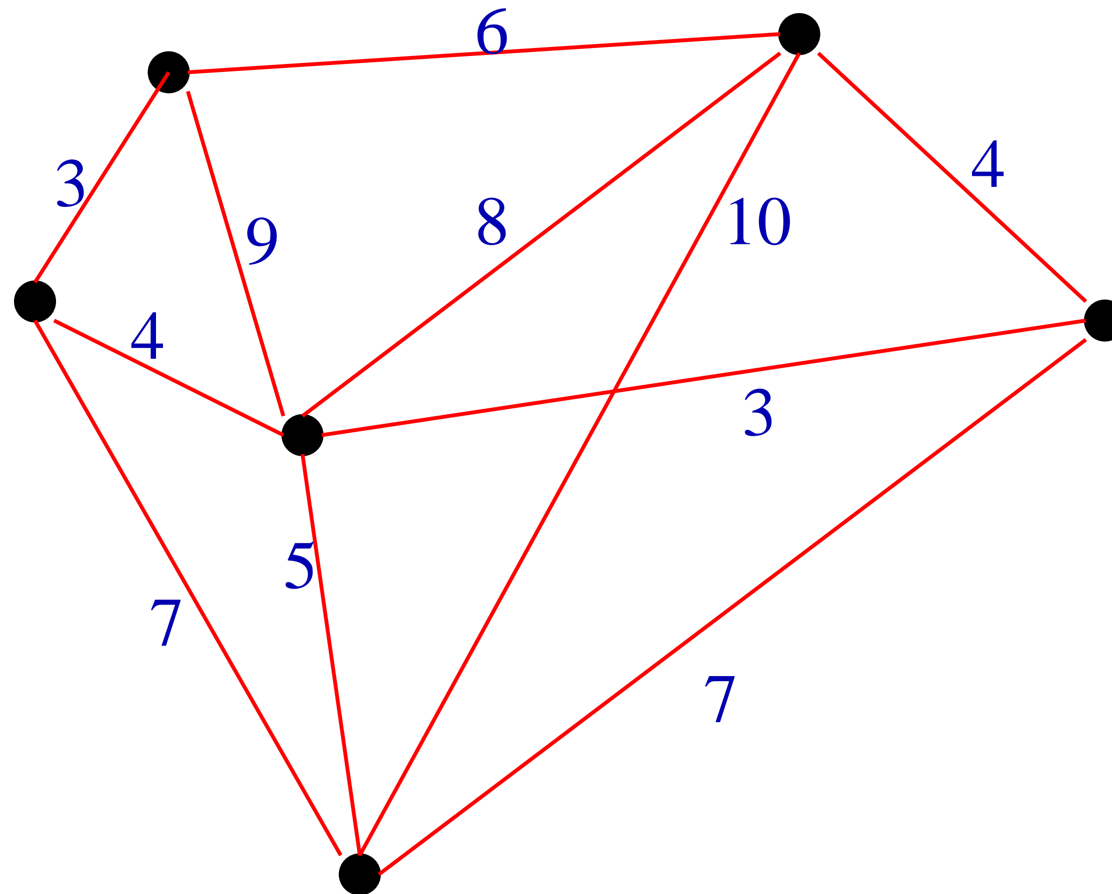
To make more practical progress we observe that memory is usually effectively infinite and model our algorithms and problem classes on this assumption.

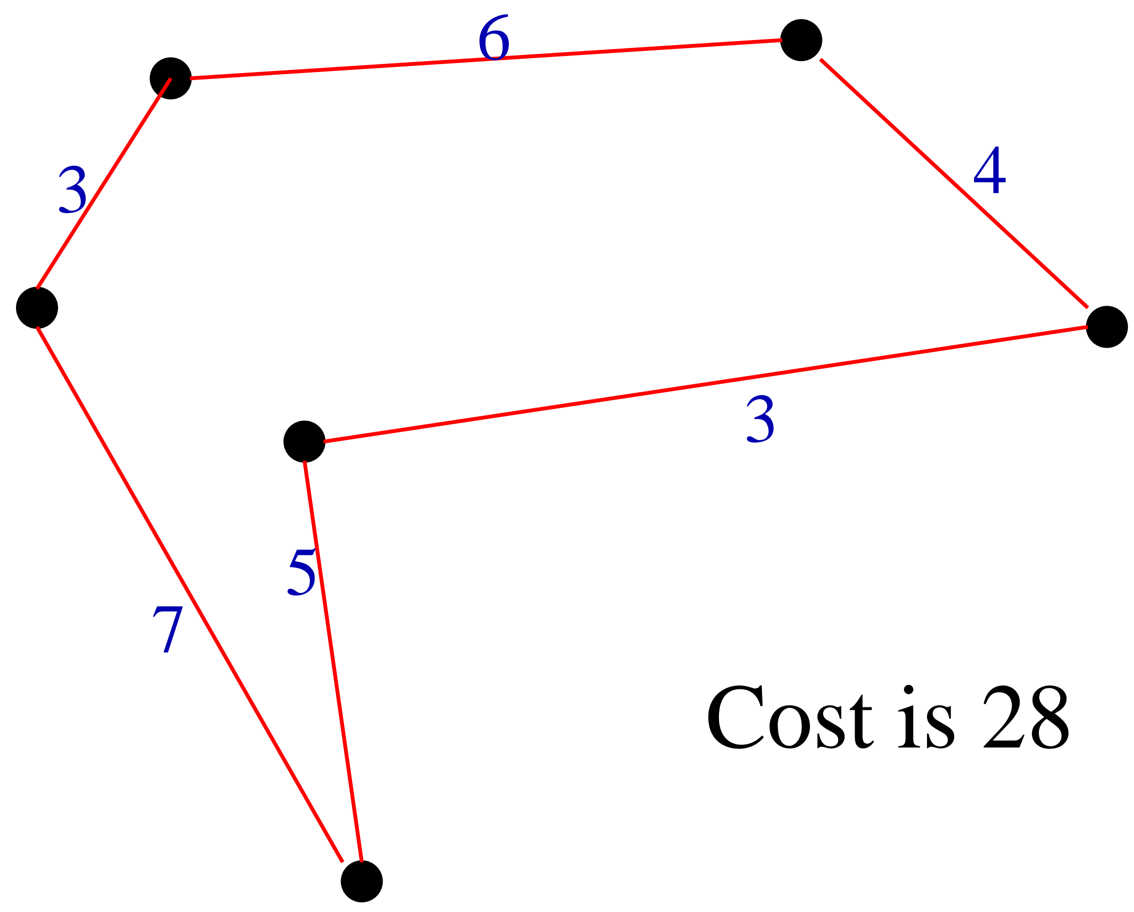
Computer Scientists group problems into several classes

- Easy problems P, which can always be solved in time which is polynomial in the size of the input (e.g. searching, sorting, multiplying matrices).
- Possibly Quite Hard problems NP, which can be solved in time no worse than exponential in the input size, but perhaps faster, in the worst case. The hardest problems in the class are called NP-Complete and include the Travelling Salesman Problem.
- Definitely Hard problems EXPTIME which always take exponential time in the worst case, including the Roadblock Problem.
- Impossible problems which are unsolvable. The best known example is the Halting Problem described by Alan Turing (1936).

Travelling Salesman Problem

- Given a set of destinations $D = \{d_1, \ldots, d_m\}$, and
- A set of roads $R = \{r_1, \ldots, r_n\}$ each joining two of the d's and each having a given length,
- Find a sequence of roads making a *tour* (i.e. that adjacent roads join up and all destinations are visited at least once) such that the overall length of the tour is the shortest possible.


6
3
9
8
10
4
4
3
5
7
7


6
4
3
3
5
7
Cost is 28

- The Travelling Salesman Problem is NP Complete.

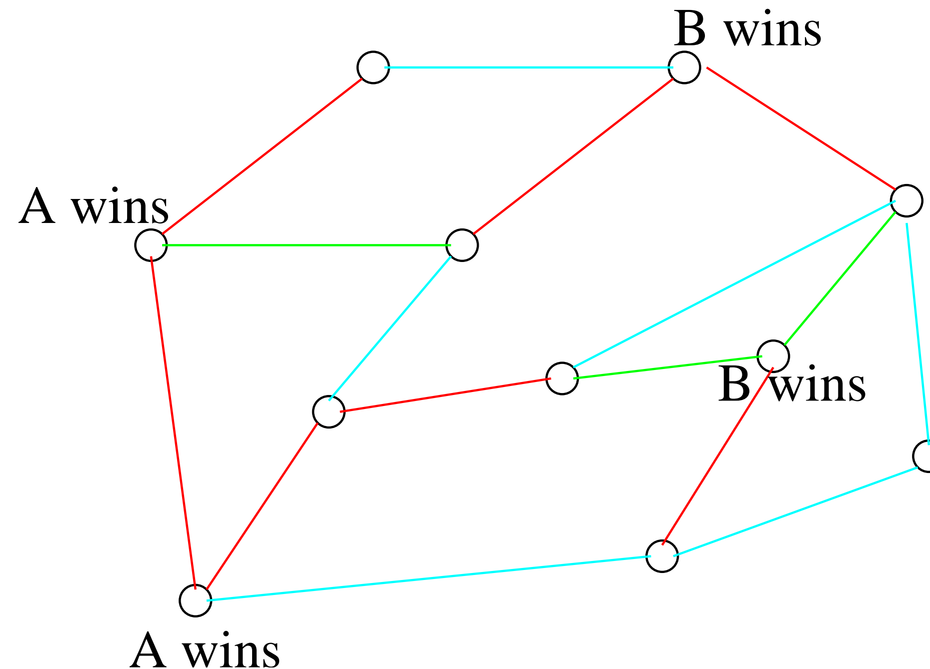
- This means that, as far as we know, there is no better way of solving it for all possible cases, than simply to examine all the possible tours and take the shortest.

- Given a map with n destinations, how many possible roads are there?

- Given a map with n destinations, how many possible tours are there?

Roadblock Problem

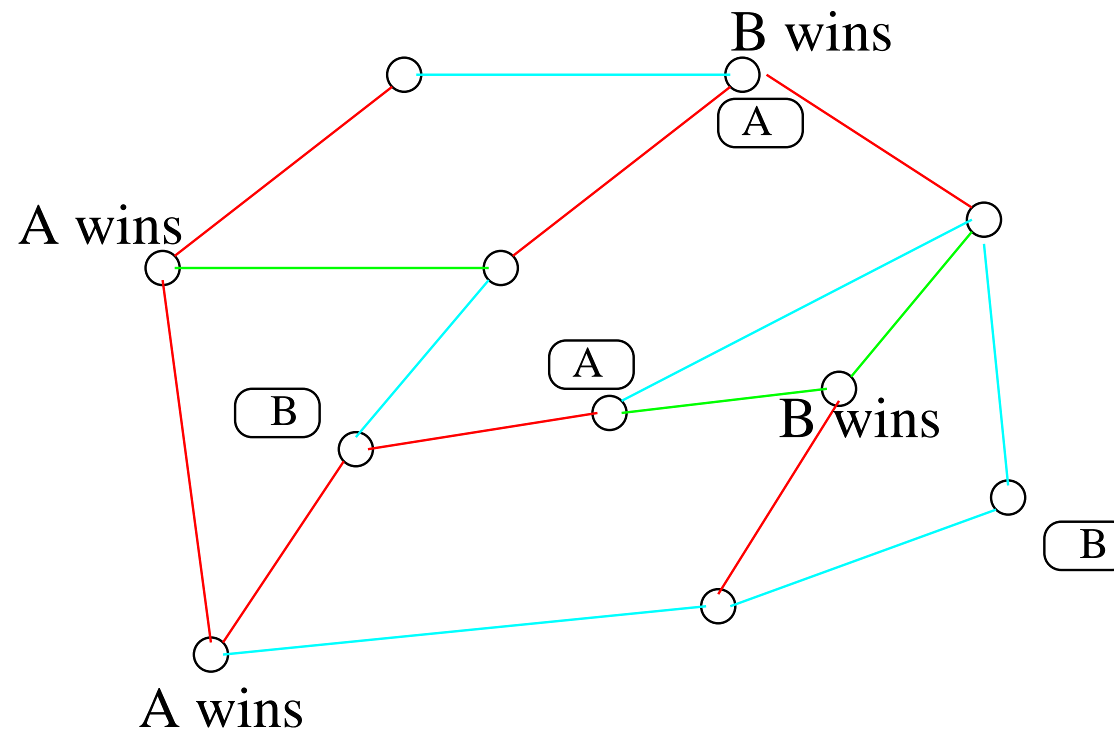
Given a road network with a collection of intersections connected by roads of three different colours. Some intersections are labelled “A wins”, some are labelled “B wins”.



Each player has a fleet of cars. The cars are distributed across the network and are located at intersections. There is at most one car at any intersection.

One move in the game is for a player to move one of their cars along a stretch of road passing a number of intersections, keeping to the same colour of road throughout. A car may not pass by or stop at an intersection that is occupied by an opponents car.

The problem is to determine if for a given starting configuration Player A (who plays first) has a winning strategy. That is there is a way for Player A to play that ensures they win regardless of the moves taken by Player B.


B wins
A
A wins
A
B
B wins
B
A wins

The Halting Problem

Parameterised by a program P and an input I, the Halting problem H(P,I) asks "Does P halt (finish) when run on I?"

This is very well defined problem. The answer is certainly either "yes" or "no" for any possible P and I.

How would you write a program to compute H?

It's no use actually trying to run P on I. If it stops we can return "yes", but what if it doesn't? We can't tell how long to wait...

In fact, the problem is impossible. No program for H can exist!

Why is it Impossible?

We use a "proof by contradiction": assume that it is possible and then show that this assumption leads us to a contradiction.

- Assume machine (program) H(P,I) solves the Halting Problem
- Use H, to build H'(X) which internally runs H(X,X) and
  - Halts if H says X(X) doesn't halt
  - Doesn't halt (infinite loop) if H says X(X) halts
- Think about what H'(H') does ....
- Construction of H' from H is sound, so our only logical conclusion is that H can't exist in the first place!

Some More Problems

Which of these problems are always solvable?

1. Given a program P and input I, how long does P take to process I?

2. Given a program P and an input I, does the program need more than 1000 bytes of data memory?

3. Given a program P, does P always halt, no matter what the input?

4. Given two programs P_1 and P_2, do they always compute the same result, given the same input?