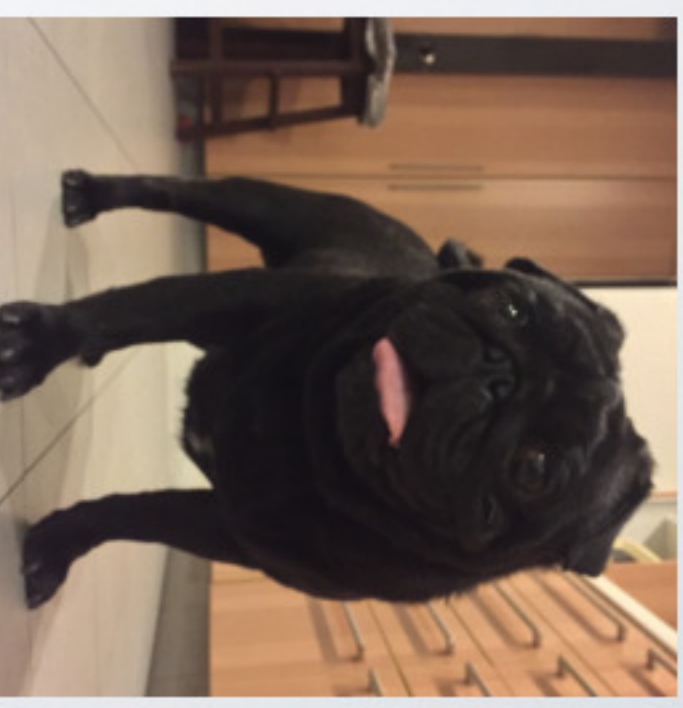# Dealing with Hard Problems

CS16: Introduction to Data Structures & Algorithms

Spring 2020

Outline

- Seating Arrangements
- Problem hardness
- P, NP, NP-Complete, NP-Hard
- Dealing with hard problems
- Problem translation
- Genetic Algorithms
- Approximations
- Travling Salesman Problem

Seating Arrangement Problem

▸ Your dating algorithms worked!

▸ You need to plan the seating arrangements for a wedding

Table 1

Table 2

Table 3

Table 4

Table 5

Table 6

Seating Arrangement Problem

- Constraints / goals

  - **k** tables

  - **n** people

  - Avoid antagonistic pairs (exes, rivals, etc) sitting at the same table

  - Maximise overall happiness

Quantifications of Pair-wise Happiness

- Assume each pair of people (A, B) has an associated 'compatibility score'

- for **friends** comp(A, B) = **10**

- for **couples** comp(A, B) = **50**

- for **antagonistic pairs** comp(A, B) = **-500**

- These values are known ahead of time

Quantifications of Table-wise Happiness

- Sum all the compatibility scores for each pair at the table

$$H\left(table\right) = \sum_{pair \in table} comp(pair)$$

Quantification of Total Happiness

- Utilitarian Approach:

$$Total\_H_{utilitarian} = \sum_{t \in tables} H(table)$$

- Egalitarian Approach:

$$Total\_H_{egalitarian} = \min_{t \in tables} H(t)$$

- Many more options!

This seems hard

- Could we just try permutations and comparing scores?

- With 60 people, 60! permutations to test

  - 8.32×10^{81}

    - ouch

- This doesn't necessarily mean that the problem *is* hard, however

Defining Problem Hardness

- Hardness of problem is defined by the runtime of the best solution

- A bad sorting algorithm *could* be $O(n!)$, but sorting in general isn't considered hard, because we have fast algorithms to solve it

- Polynomial Runtimes
  - $O(n)$, $O(n^2)$, $O(n^{500})$
  - Problems with these solutions are **tractable**

- Super-Polynomial Runtimes
  - $O(n!)$, $O(2^n)$, $O(n^n)$
  - Problems with these solutions are **intractable**

58

Exponential vs. Polynomial Growth Rates

log(y)

x

n^n
n!
n^{100}
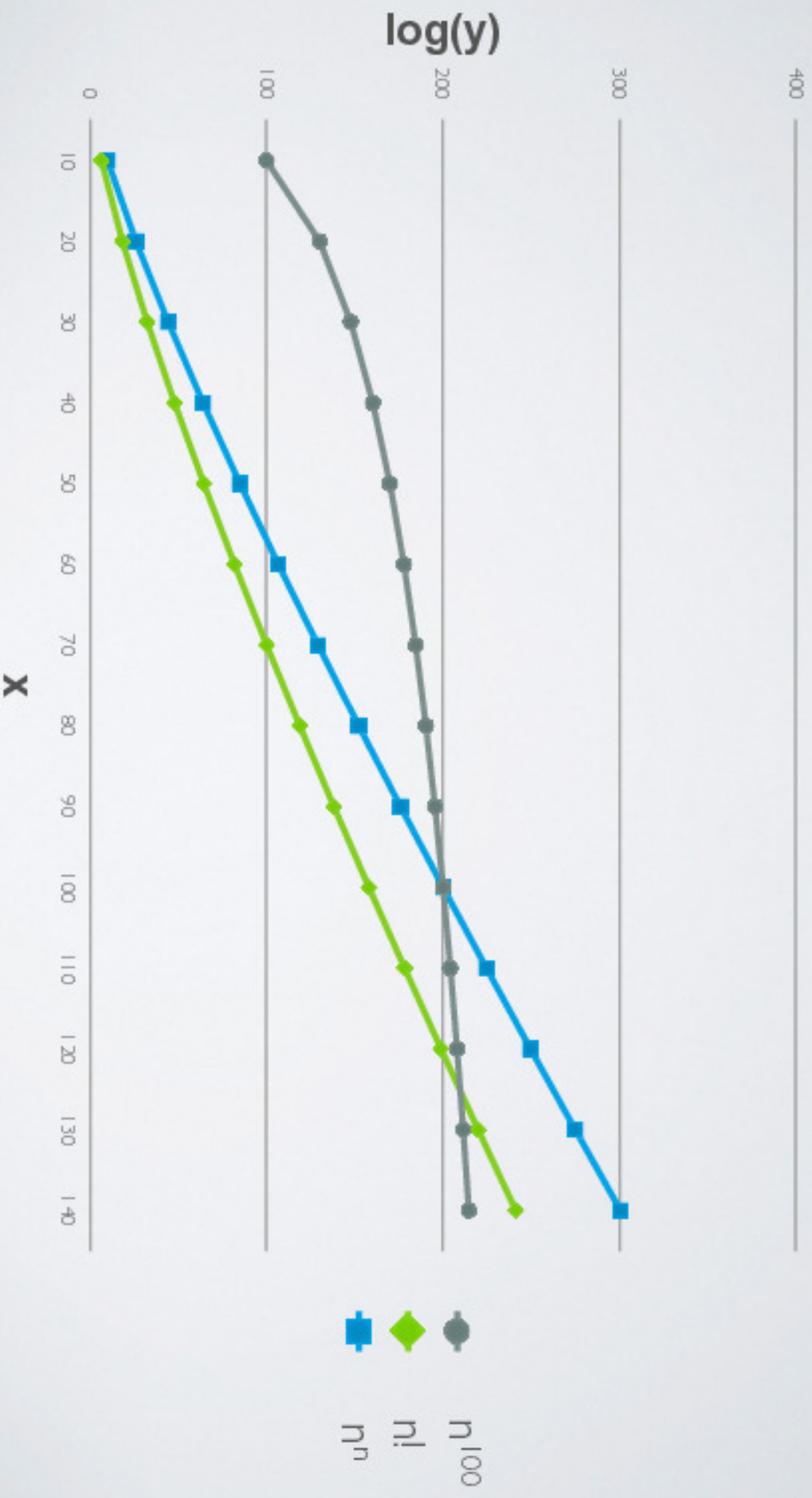
Categories of Hardness

- NP
  - The set of problems for which we can verify the correctness of a solution in polynomial time

- P
  - A subset of NP, where the problem is solvable in polynomial time

- NP-Complete
  - "The hardest problems in NP"
  - Solution is checkable in polynomial time
  - not known whether there exist any polynomial time algorithms to solve them

- NP-Hard
  - Problems that are "at least as hard as the hardest problems in NP"
  - Don't necessarily have solutions that are checkable in polynomial time

NP-Hard

NP-Complete

NP

P

Complexity
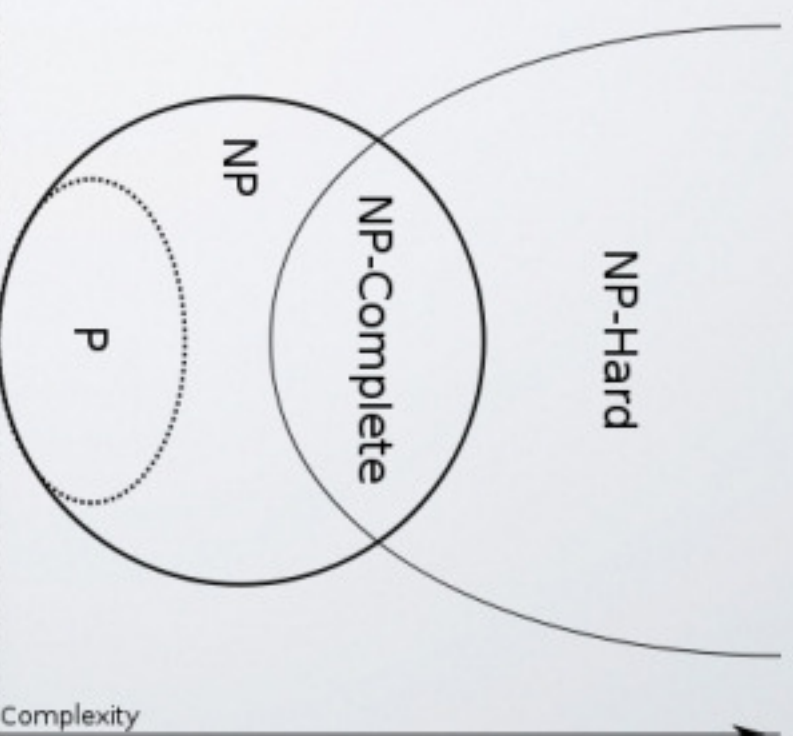
Back to our seating arrangement

- To get an intuition as to how hard our problem is, let's see if we can convert it into a problem that has already been proven to be in NP, P, NP-Complete, or NP-Hard

- But… where to start?

Constraint Relaxation

- See if you can solve an 'easier' version of the problem, by removing some of the properties that make the problem hard

- In real life
  - "what would you do if you could not fail?"
  - "which job would you take if they all paid equally?"
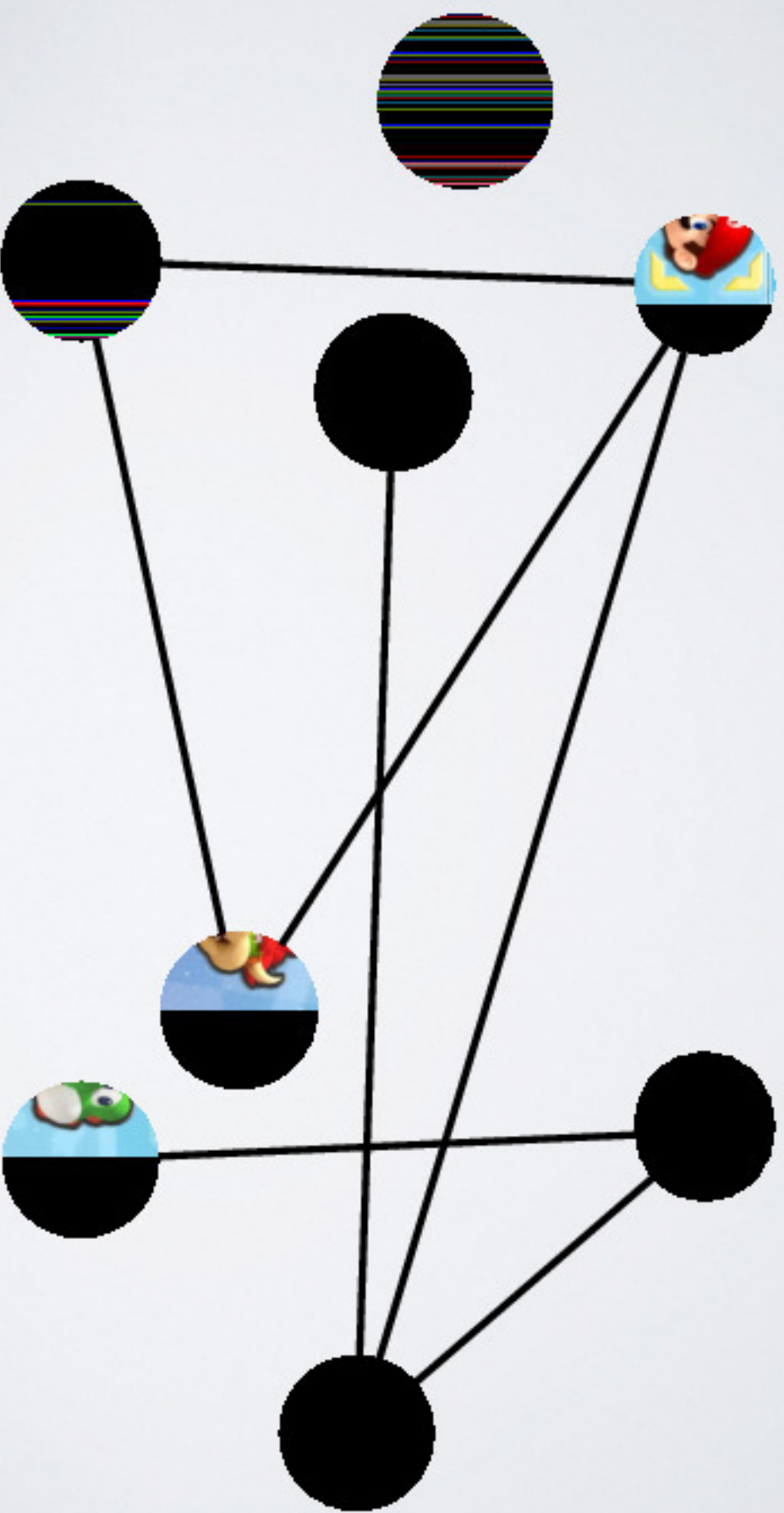
Let's avoid disaster

- Constraints / goals

  - \# of tables

  - \# of people

  - Avoid antagonistic pairs (exes, rivals, etc)

  - ~~Maximise overall happiness~~

- Hopefully, having no tables with antagonistic pairs will put in the right direction for maximising overall happiness

Relationships as a graph

- edge key:
  - **friends**
  - **couple**
  - **enemies**
  - no edge = no prefs

64

An Antagonism graph

65
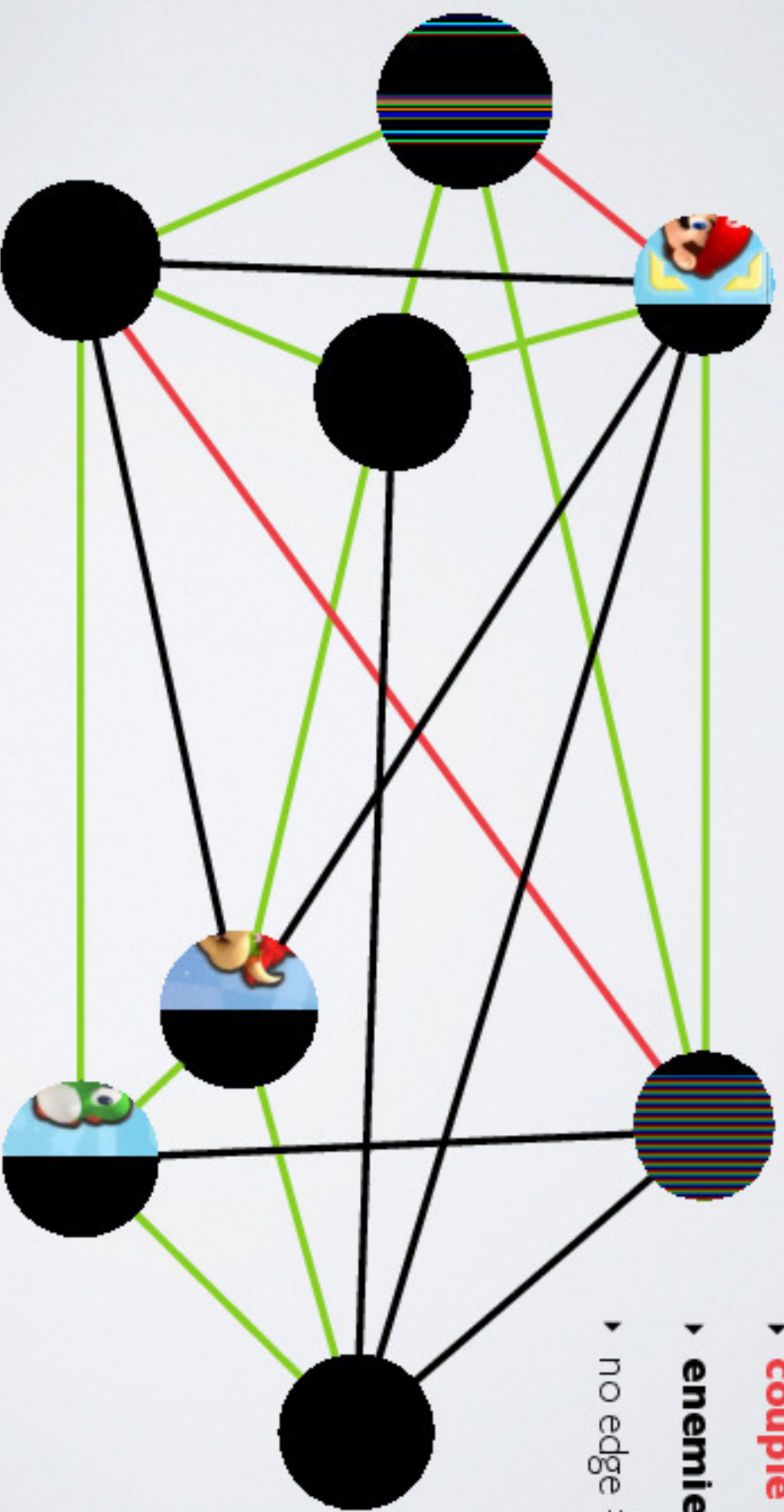
Translating the problem

- Now, we have these antagonistic relationships represented as a graph!

- Question is no longer:

  - Can we avoid antagonistic pairs (exes, rivals, etc) sitting at the same table, given **n** people and **k** tables?

- Instead:

  - Use colours to represent different tables, so:

  - Could we assign 1 of **k** colours to each node in the antagonism graph, such that no two nodes that share an edge have the same colour?

An Antagonism graph
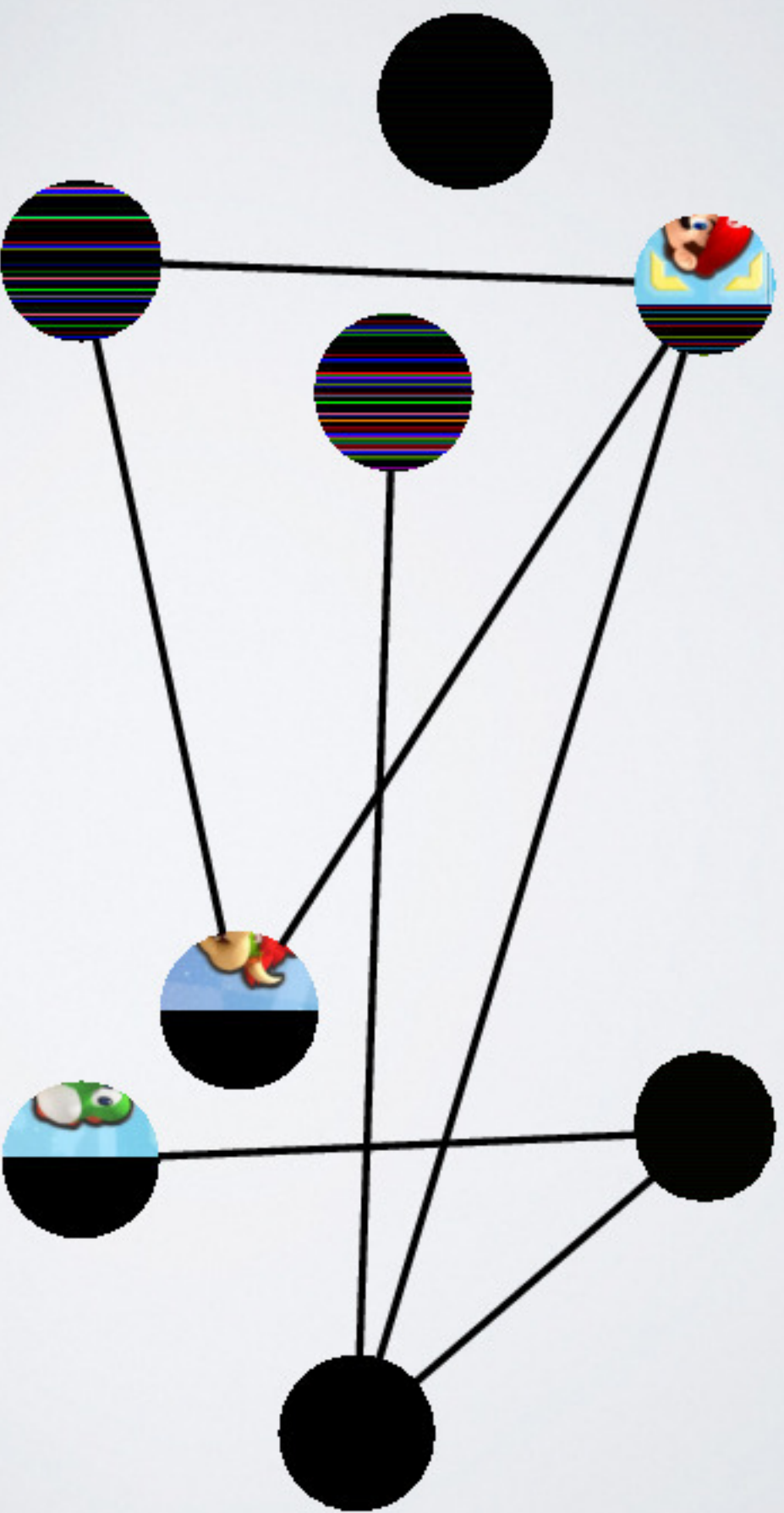
Lecture Activity 3

Try out the Graph k-colouring problem!

2 Mins......

Lecture Activity 3

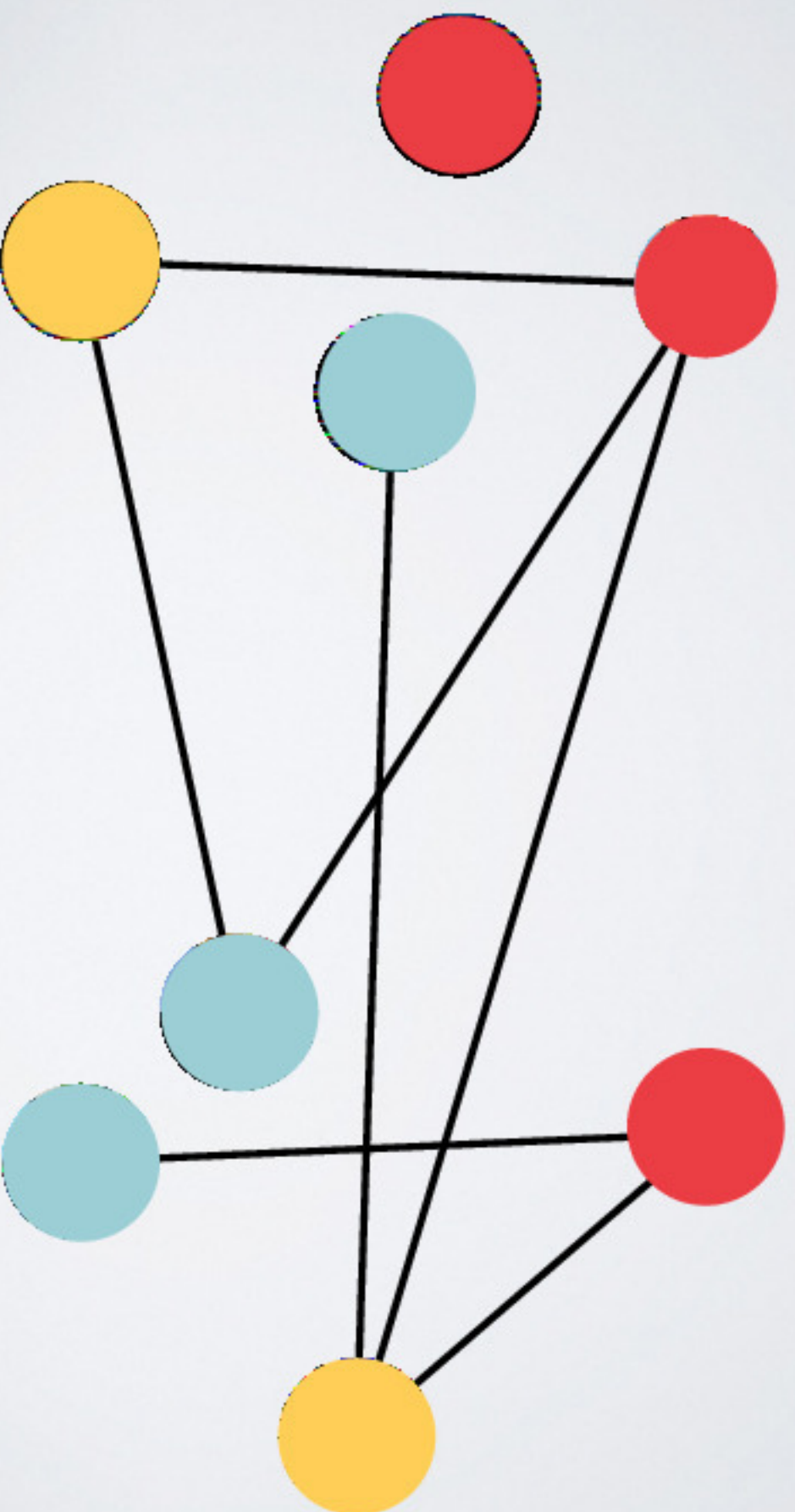Try out the Graph k-colouring problem!

I Min.....

Lecture Activity 3

Try out the Graph k-colouring problem!
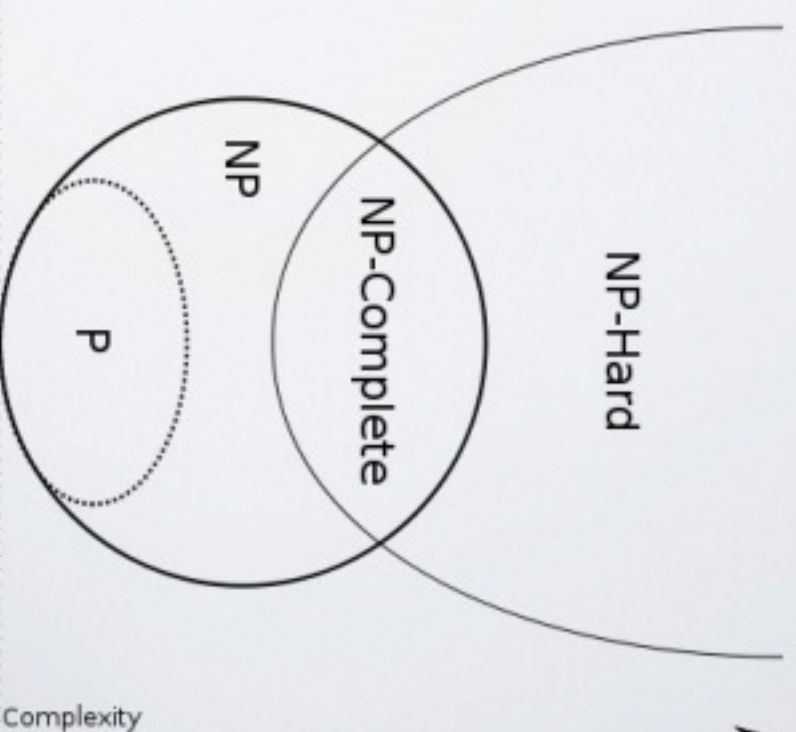
0 Mins......

Lecture Activity 3 Answers

Answers!

Graph k-colouring

- Generally, the problem of determining whether nodes in a graph can be coloured using up to **k** separate colours, such that no two adjacent vertices share a colour

- This is NP-Complete!

- And thus, even this much easier version of the problem is **very hard**

NP-Hard

NP-Complete

NP

P

Complexity →

Are we screwed?

- The best algorithms to solve the graph k-colorability problem take $O(2.445^n)$ time and space

- With 60 guests, $2.445^{60} = \sim 450$ billion

  - which isn't that bad

  - Modern computers can handle ~ 3 billion 'operations' / sec, so this would take more than a couple minutes, probably less than 15

- But we've still only avoided the worst case!

Genetic Algorithms

- A form of 'guess and check', using a number of possible solutions to a problem

- Inspired the process of evolution

Biology Review

- All organisms are made up of **genes**, where genes (or a combination many genes) interact to produce our **phenotype**, the expression of those genes

- We are all a combination of a mix of our parents genes, and some random mutations
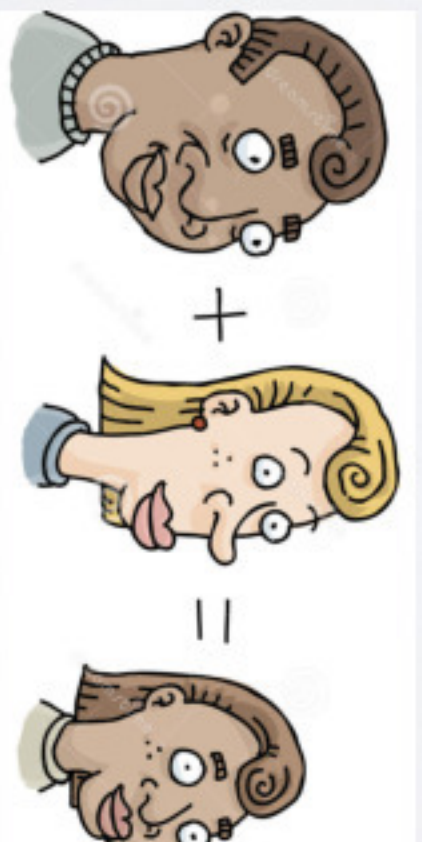
Evolution via Sexual Reproduction, broadly

- There exist an initial population of organisms within a species

- The 'sexually fit' organisms reproduce

  - Take some genes of parent A, some of parent B

  - add some random noise

  - this new collection of genes is a new specimen, AB'

- Older + less fit parts of populations die off, leaving the survivors to repeat the reproduction process

Solution Mating

Total_H = 300

+

Total_H = 325

Total_H = 400

78

High-Level Genetic Algorithm Pseudocode

```
function geneticAlgo(opt_seed_sols):
    solution_set = opt_seed_sols or || randomly generated initial population of solutions
    init_size = size(solution_set, threshold, time_limit)

    while True:

        new_gen = []
        for some number of iterations:

            A, B = 2 solutions from solution_set, drawn at random
            AB' = a new solution that combines properties of A and B
            randomlyMutate(AB')
            new_gen.append(AB')
        solution_set.addAll(new_gen)
        rank solutions in solution_set based on 'fitness'
        remove all but init_size many best solutions from solution_set
        if best(solution_set) > threshold or time_limit has passed:
            break
    return highest ranking solution from solution_set
```

Genetic Algorithms

- If seeded with 'good' solutions for the initial population of solutions, output is guaranteed to be at least as good as the best of the initial solutions

- Can come up with unexpected solutions

- Tend to do really well!