CHES Algorithmics Orientation

5th December 2023

A. Graph Models

B. Graph Models Student Activity

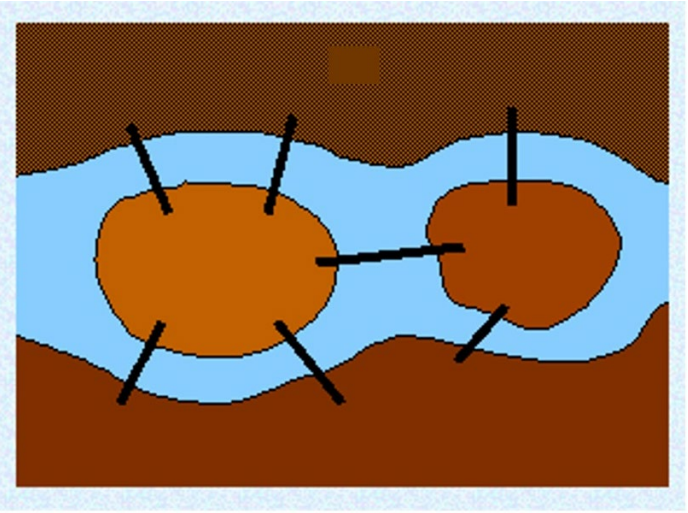
C. Defining Algorithms

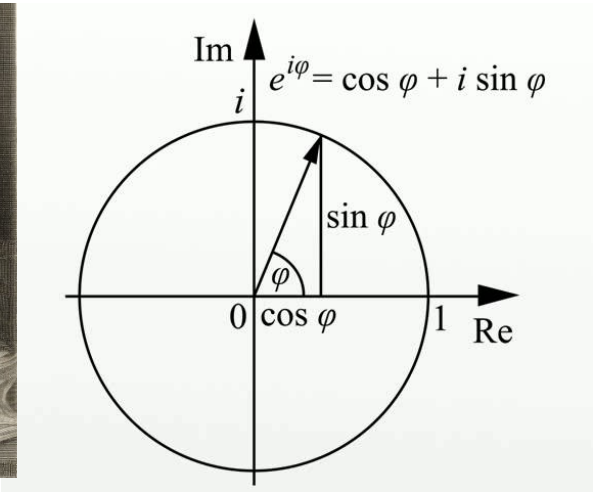
D. Defining Algorithms Holiday Homework

A. GRAPH MODELS

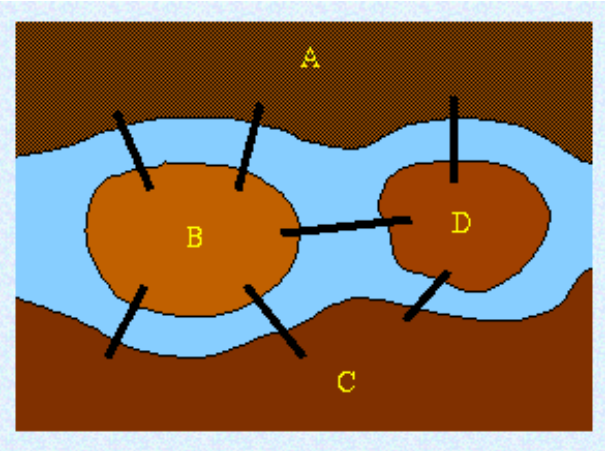
The Bridges of Königsberg

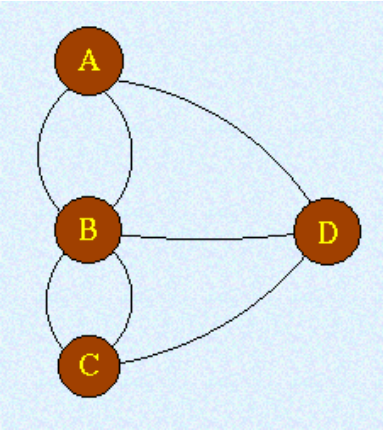
- The Prussian city of Königsberg spanned both banks of a river with two islands in it. Seven bridges connected both banks and both islands with each other.
- How to walk across both banks and both islands by crossing each of the seven bridges only once?
- Around 1736, this problem was sent to the most prolific and famous mathematician of the day **Leonard Euler.**





The Bridges of Konigsberg



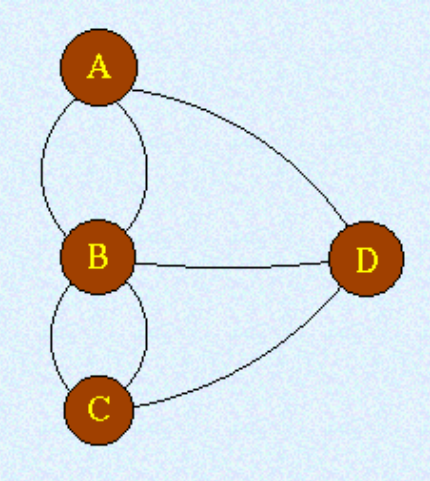


Euler was intrigued with this seemingly trivial problem and his great innovation was in viewing the problem **abstractly** by using lines and letters to represent the landmasses and bridges.

A completely new type of thinking for the time and the birth of a new branch of mathematics called **graph theory.**

In graph theory, nodes and edges make up graph models that represent abstractions of real world entities.

The Bridges of Konigsberg

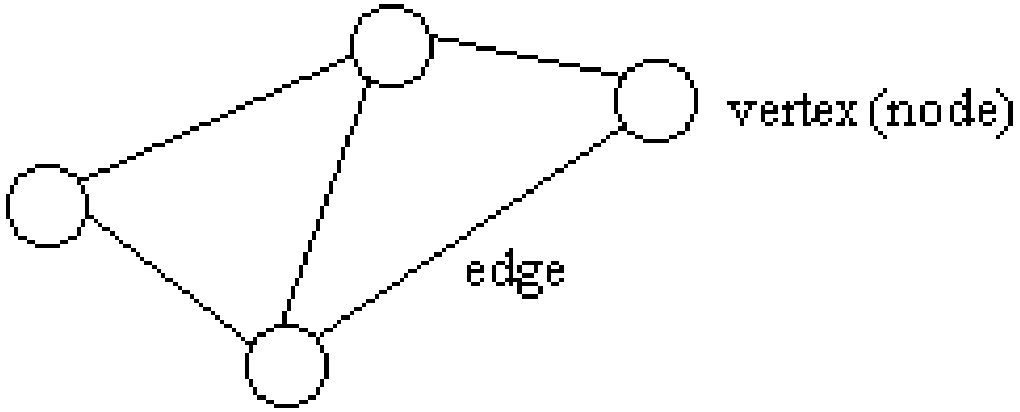


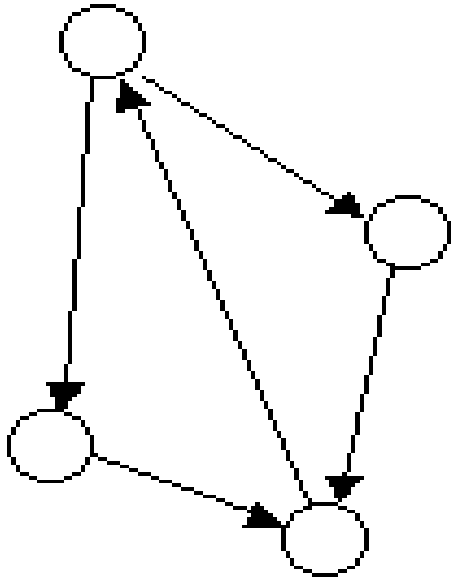
How to walk across both banks and both islands by crossing each of the seven bridges only once?
Euler's Abstraction – labelled circles for land mass locations, with lines for connections.

Even number of connections to a land mass location allow for equal enters/exits, but odd numbers of connections are more difficult to solve.

Euler proved that the Bridges Problem could only be solved when there are either 0 or 2 locations with odd-numbered connections (degree), and if the path starts at one of these odd-numbered connections, and ends at another one.

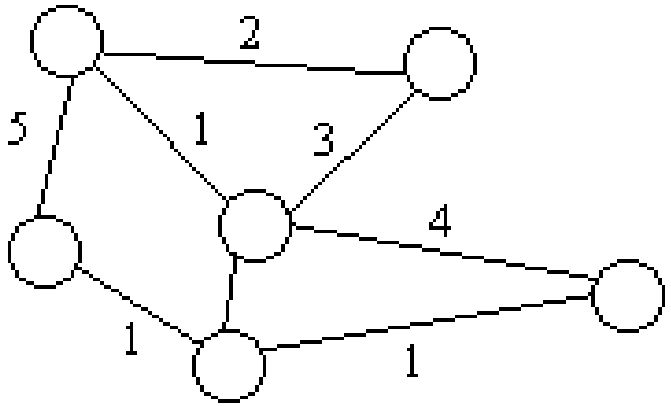
Königsberg had all four landmasses with odd connections, and so there is no path that crosses each of the seven bridges only once.





directed graph

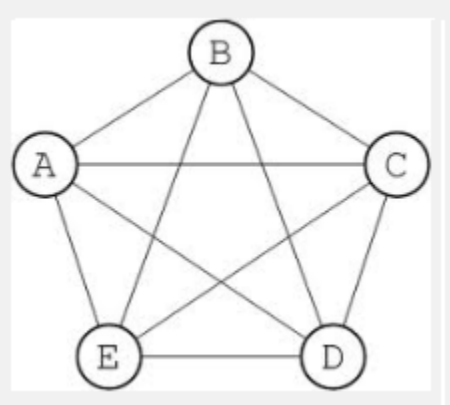
Graph Theory

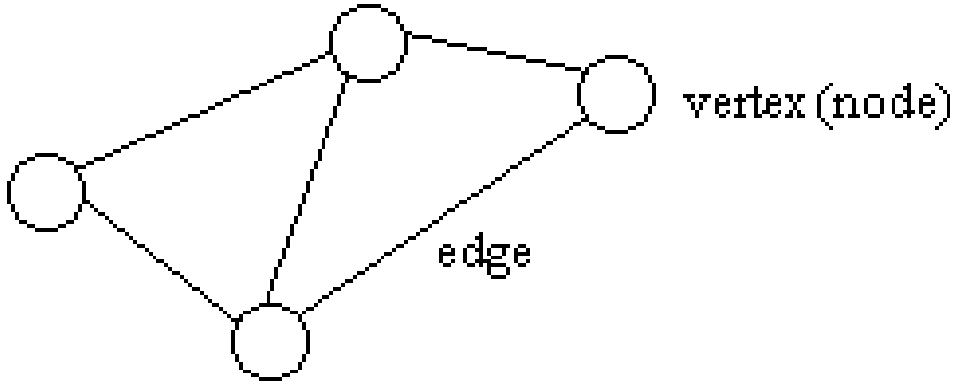


weighted graph

Graph Theory

- A ***graph*** is made up of ***vertices (nodes)*** usually represented by circles with or without labels and lines called ***edges*** that connect them. Represented as sets G={V,E}

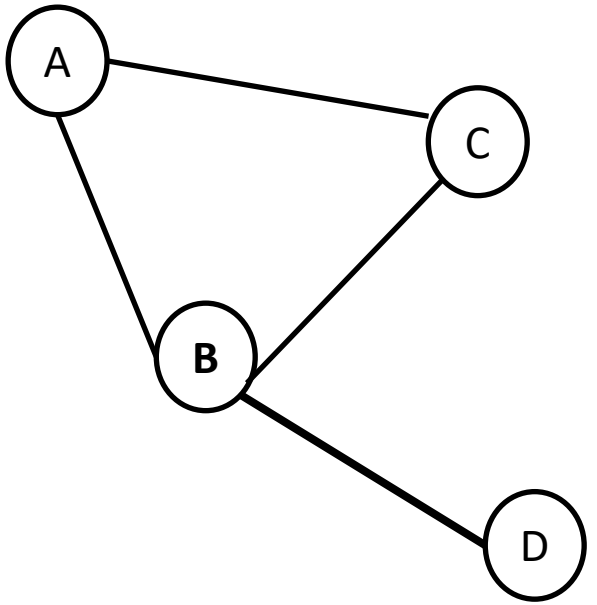




- Graph theory should not be confused with the graphs of functions.

Undirected graphs

- ***Undirected graphs*** have edges that can be travelled along in any direction.
- The ***degree*** of a vertex is the number of edges that come off it.

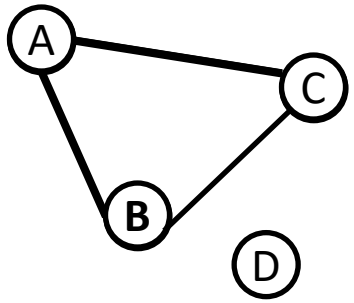


G={V,E}
V={A,B,C,D}
E={A-B, A-C, B-C, B-D}

Deg(A)=2,
Deg(B)=2,
Deg(C)=2,
Deg(D)=1

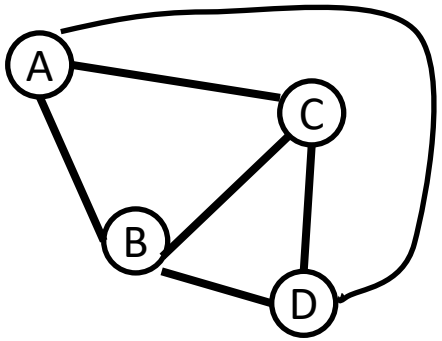
Undirected graphs

- An ***Isolated vertex*** is not joined by any edges.
- A graph is connected when each vertex can be reached from any other vertex.
- ***A subgraph*** is made up of a subset of the original graph.
- A ***complete graph*** has every pair of vertices joined by one edge.



Deg(A)=2,
Deg(B)=2,
Deg(C)=2,
Deg(D)=0

Isolated Vertex D



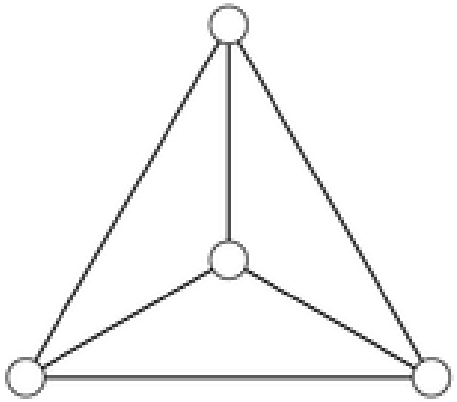
Deg(A)=3,
Deg(B)=3,
Deg(C)=3,
Deg(D)=3

Complete Graph

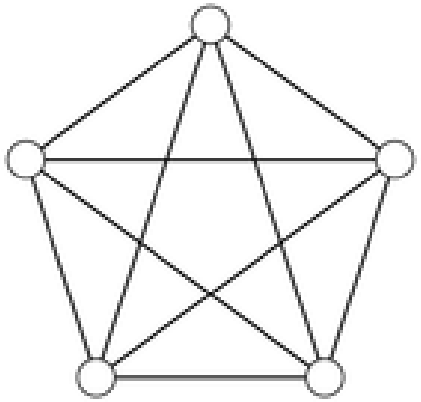
Undirected graphs

- A ***complete graph*** has every pair of vertices joined by one edge.

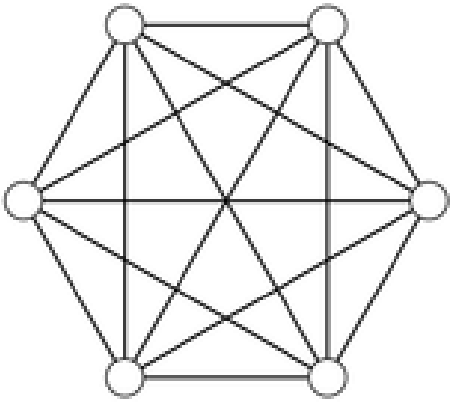
- A complete graph with "n" vertices, each vertex is of $(n-1)$ degrees, and there are $\frac{n(n-1)}{2}$ edges.



K_4



K_5



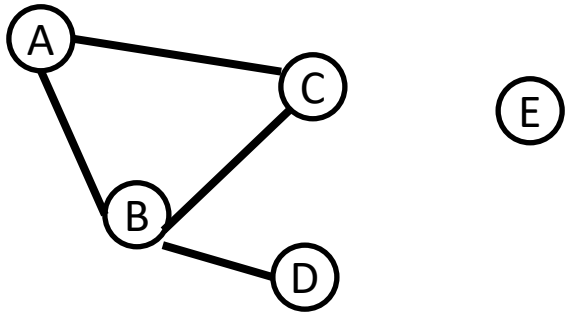
K_6

3 Graph Topology representations

Adjacency Matrix :

1 indicates an edge, 0 no edge

	A	B	C	D	E
A	0	1	1	0	0
B	1	0	1	1	0
C	1	1	0	0	0
D	0	1	0	0	0
E	0	0	0	0	0



G={V,E}

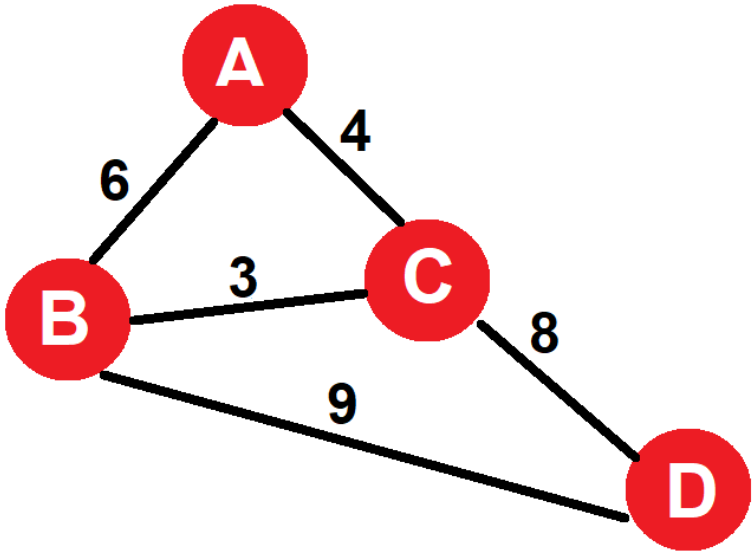
V={A,B,C,D,E}

E={A-B, A-C, B-C, B-D}

Weighted Graphs

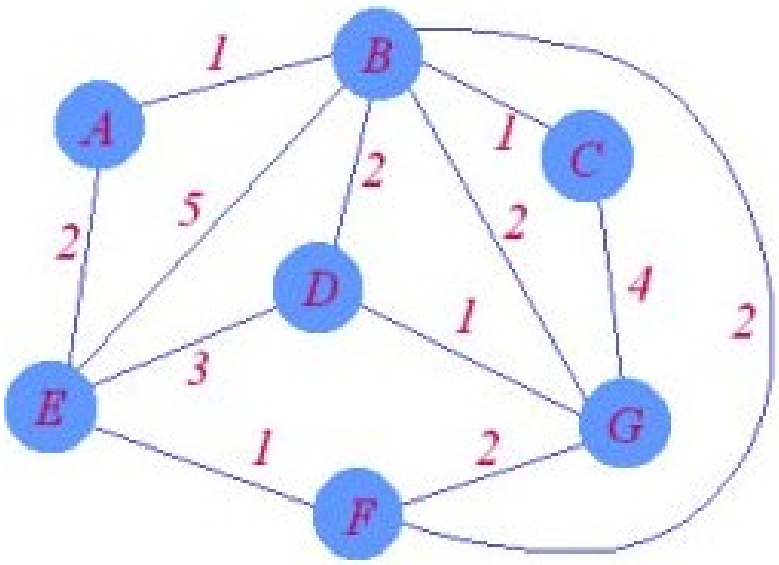
- A weighting is assigned to each edge representing a quantity such as cost or distance or other type of proximity or relationship.

G={V,E}
V={A,B,C,D}
E={A-B, A-C, B-C, B-D, C-D}
w(E)={6, 4, 3, 9, 8}



Example: Weighted Graphs

- A weighting is assigned to each edge
- A matrix representation is also shown of this weighted information

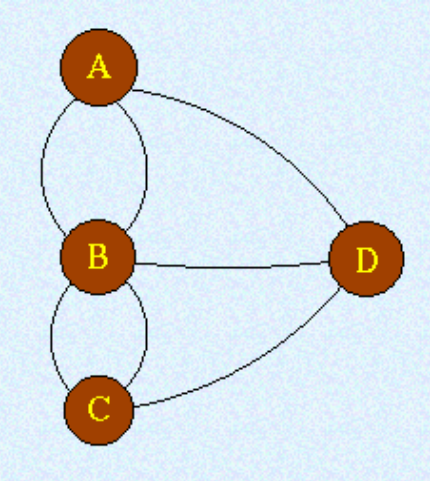


edge	*list*
AB	1
AE	2
BC	1
BD	2
BE	5
BF	2
BG	2
CG	4
DE	3
DG	1
EF	1
FG	2

matrix

	A	B	C	D	E	F	G
A	-	1	-	-	2	-	-
B	1	-	1	2	5	2	2
C	-	1	-	-	-	-	4
D	-	2	-	-	3	-	1
E	2	5	-	3	-	1	-
F	-	2	-	-	1	-	2
G	-	2	4	1	-	2	-

The Bridges of Konigsberg



How to walk across both banks and both islands by crossing each of the seven bridges only once?

Even number of connections to a location allow for equal enters/exits, but odd numbers of connections are more difficult to solve.

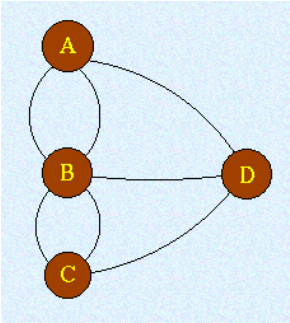
Euler proved that the Bridges Problem could only be solved if the entire graph has either 0 or 2 nodes with odd-numbered connections (degree), and if the path starts at one of these odd-numbered connections, and ends at another one.

Königsberg has four nodes of odd degree, and so there is no path that crosses each of the seven bridges only once.

Traversable Graphs

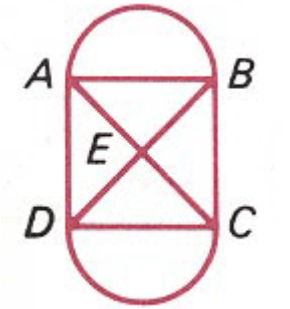
a) **No Euler Path/Circuit** if a graph has ***more than two vertices of odd degree.***

a)



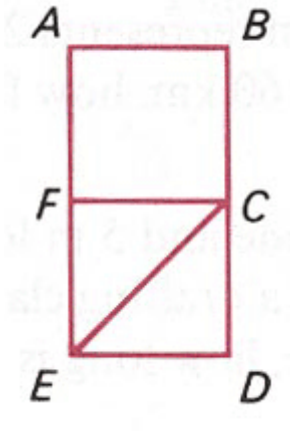
b) **Euler Path/Circuit exists** if all vertices of a graph have ***even*** degree, that is zero nodes of odd degree.

b)



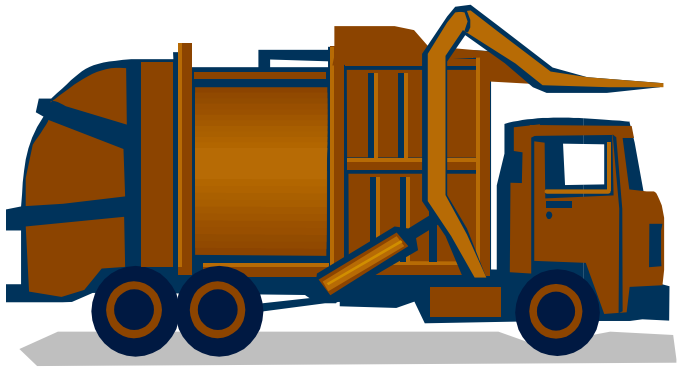
c) **Euler Path exists** if a connected graph has ***exactly two odd*** vertices. The starting point must be one of the odd vertices and the ending point will be the other of the odd vertices.

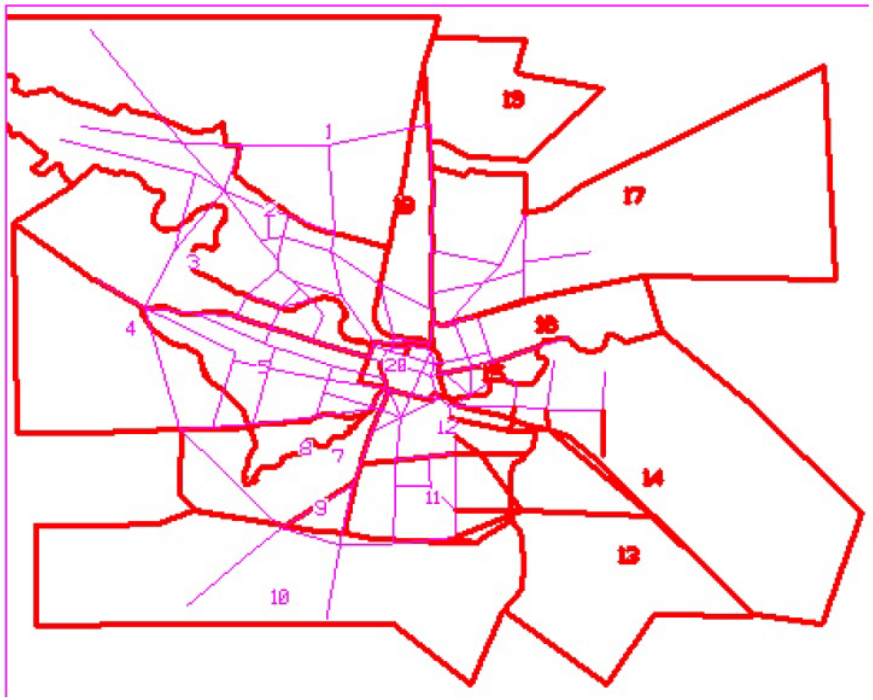
c)



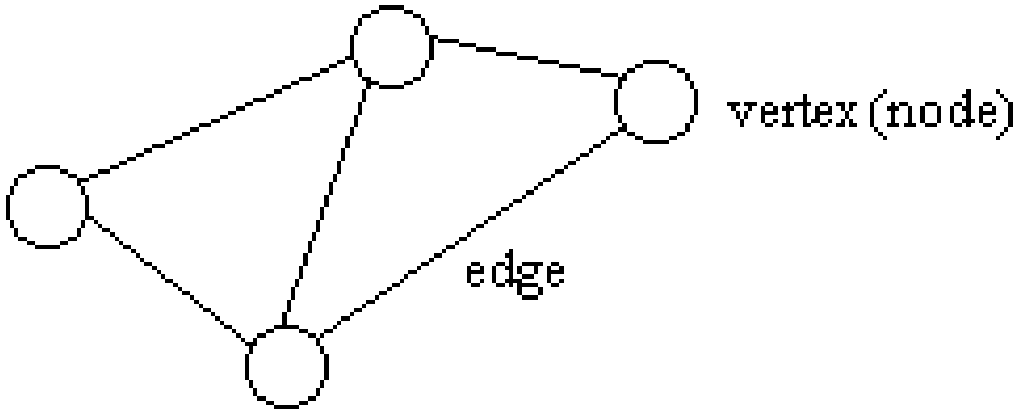
Euler Path/Circuit

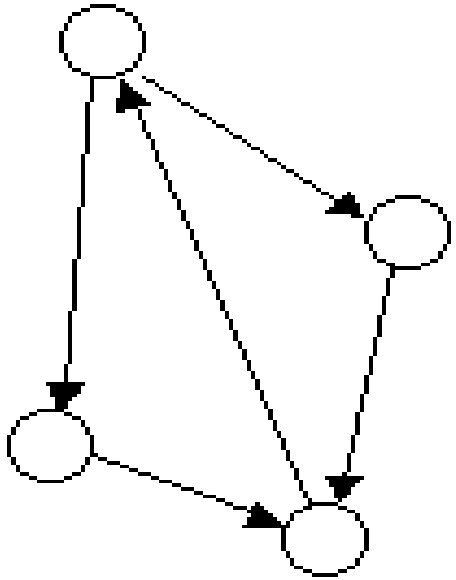
An ideal Rubbish Truck collection route is an Euler Circuit as the truck most efficiently will travel every street once only to collect the rubbish before returning to the depot.





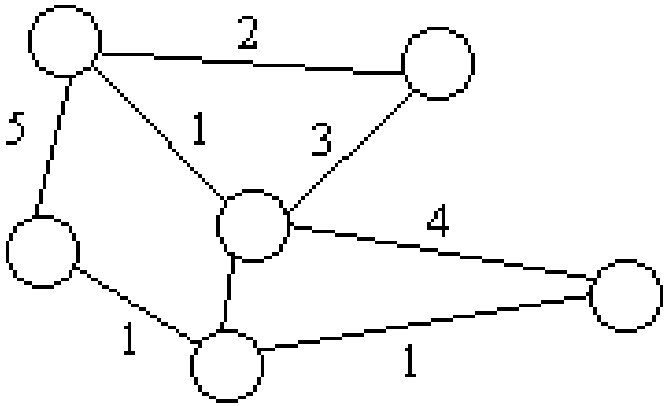
B. GRAPH MODELS STUDENT ACTIVITY





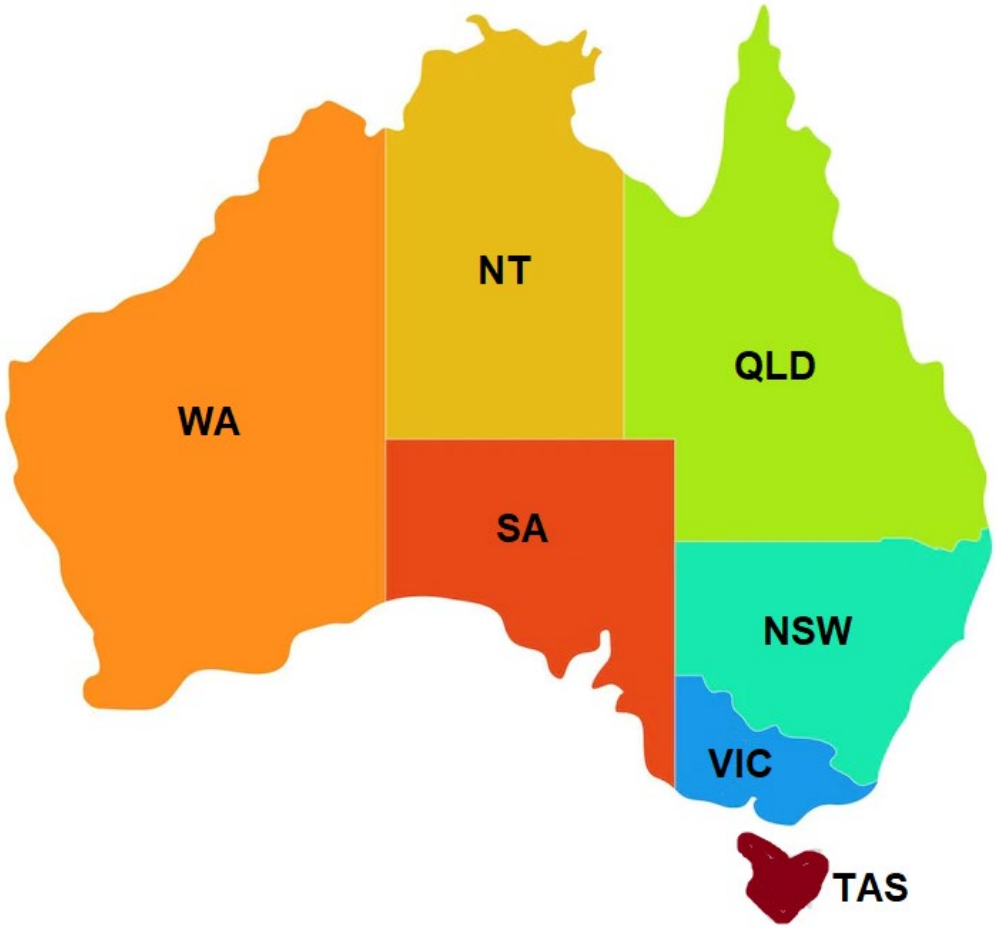
directed graph

Graph Model Abstractions



weighted graph

A. Modelling information with Graphs

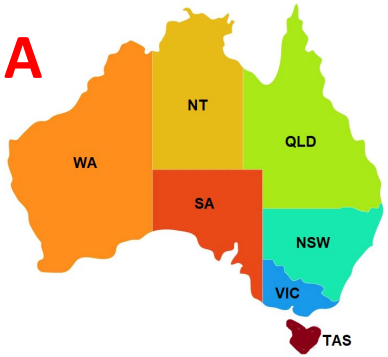


B. Modelling information with Graphs

C. Modelling information with Graphs

Modelling information with Graphs

A



B

C

Blue Focus

A. Representing geographic regions sharing borders
B. Representing distances to travel between locations
C. Representing emails sent between co-workers in a week

Red Focus

A. Representing people migration numbers between regions
B. Representing traffic congestion between locations
C. Representing compatibility between a group of people

Modelling information with Graphs

Question 12 (11 marks)

HexaReverso is a single-player game played with hexagonal tiles. Each tile has an integer value, with one side having a positive value and the reverse side having the negative of that value. The tiles are arranged in a hexagonal shape. Large hexagonal grids can be hundreds of tiles wide.

Modelling information with Graphs

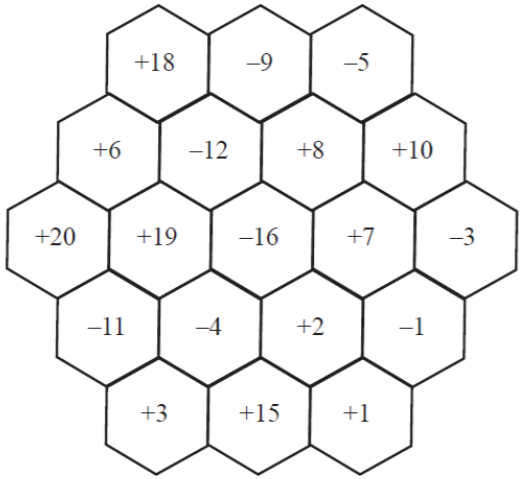
Question 12 (11 marks)

HexaReverso is a single-player game played with hexagonal tiles. Each tile has an integer value, with one side having a positive value and the reverse side having the negative of that value. The tiles are arranged in a hexagonal shape. Large hexagonal grids can be hundreds of tiles wide.

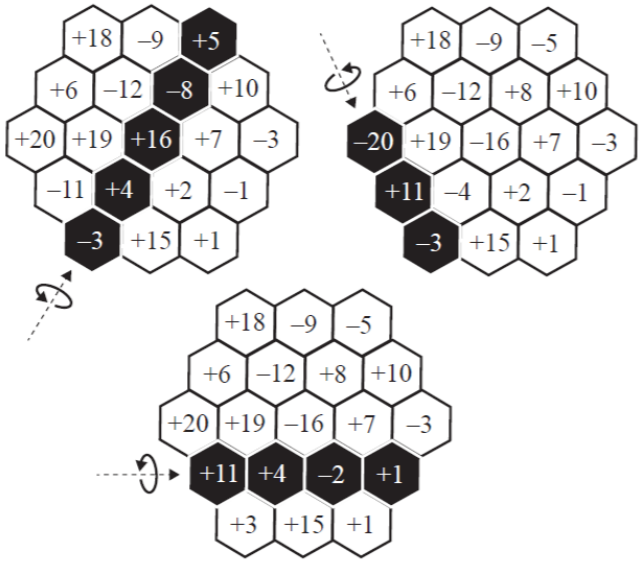
The goal of the game is to maximise the sum of the face-up values on the tiles.

A 'row' refers to a straight line of sequentially adjacent tiles. In the game, a player may flip any row of tiles to its reverse side. Three examples of this are shown in the diagram below. The player may flip any number of rows. It is known that this flip operation does not allow for all possible grid arrangements to be generated.

An example of a small HexaReverso board



Three examples of the flip operation



a. It is important that tiles in a particular row can be efficiently identified.

i. Describe how data about the tiles in a HexaReverso game could be stored. A single ADT or a combination of ADTs may be used. 3 marks

D. DEFINING ALGORITHMS

defining algorithms

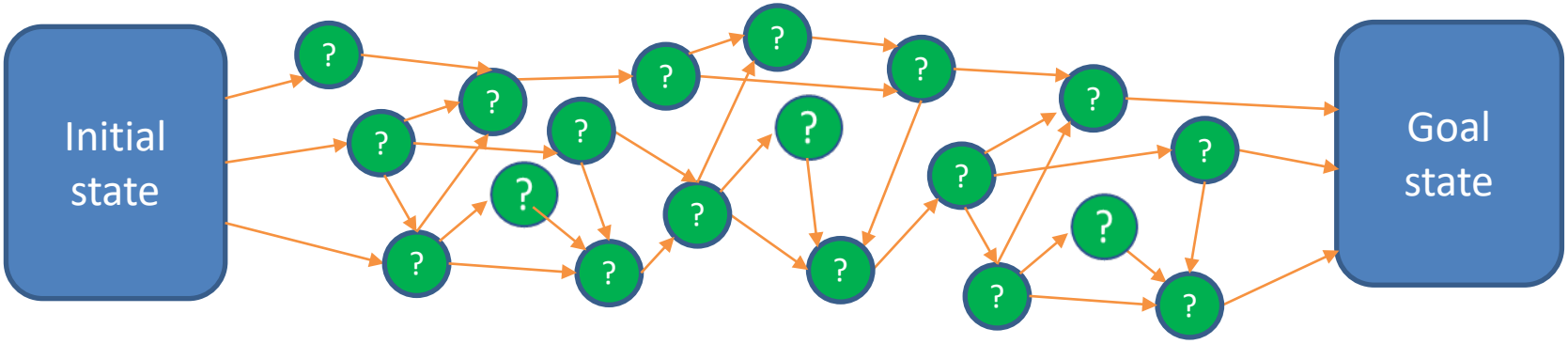
What's an algorithm?

An **algorithm** is a **set of instructions** to be **followed systematically,** that defines a general process for proceeding from an **initial state** to a final **goal state** which may solve a problem or create an object or resource.

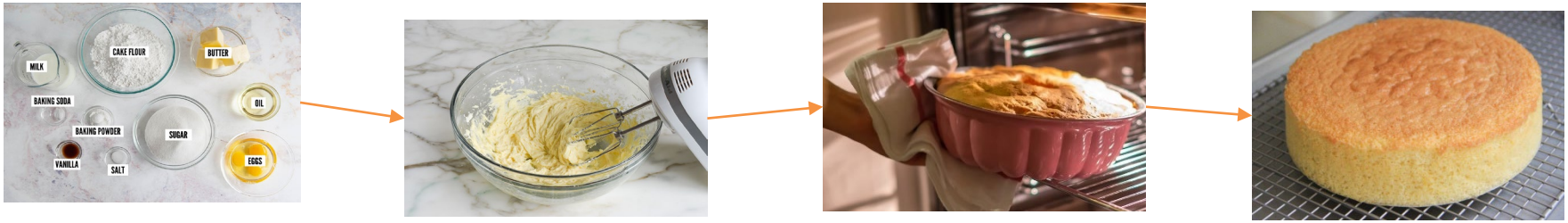
For example a **recipe** is a kind of algorithm that starts with an **initial state** of assembling the **ingredients**, cooking tools and appliances and **stepping through** to the final **goal state** of creating **servings of food**.

How does an algorithm work?

- As an algorithm proceeds, we can think of it as moving from the original **state** to the **goal** or solution state by stepping through a sequence of states within what we might call the '**problem space**' for the algorithm.



- For example to bake a cake, we move from separate **ingredients** (butter, milk, sugar, eggs flour); through the **stages** of **combining the ingredients** in a cake-batter; to the cake batter in a cake tin being heated in an oven to the **finished cake**.



How do you create an algorithm?

Given a particular kind of problem, we start by **imagining** the **problem space** for that problem: the states an algorithm will need to **step through** in order to get from the **inputs** to the **goal** or solution.

There are various levels of algorithm success we might achieve:

1) Our algorithm **reliably arrives** at a solution state on (most) inputs.

2) Our algorithm **efficiently** arrives at a solution state on (most) inputs.

3) Our algorithm arrives in **the most efficient** way at a solution state on (most inputs).

Showing that an algorithm has achieved **(3)** is generally difficult and is something we will look at in more detail in Unit 4.

In this session, we will start with some general approaches to writing algorithms, which can get you to **(1)** and, maybe **(2)**.

Exhaustive searching

Systematically **create every possible state** in the problem space, and **check** each state option to see if it is a solution that satisfies the criteria.

(A exhaustive searching approach to designing a cake recipe that just tells you to throw **all the possible ingredients** for a cake **together in all different ways (combinations)**, cooking them at **different points** in the process for **different lengths of time**, at **different temperatures**, then **check the outcome** in each case to see if you have succeeded yet in **making a cake**!)

The Exhaustive searching approach is also sometimes known as '**generate and test**' or '**brute force**' approach.

Minimum coins for change example

An algorithm for giving the **denominations** for the **minimum** number of coins required to make a certain amount of change. So **input** is change amount, **output** are the coins required.

Suppose the coin denominations in our currency are {**1, 3, 4**} cent pieces.

Then, for *e.g.* for **19¢** change we would give **four 4¢** and a **3¢**

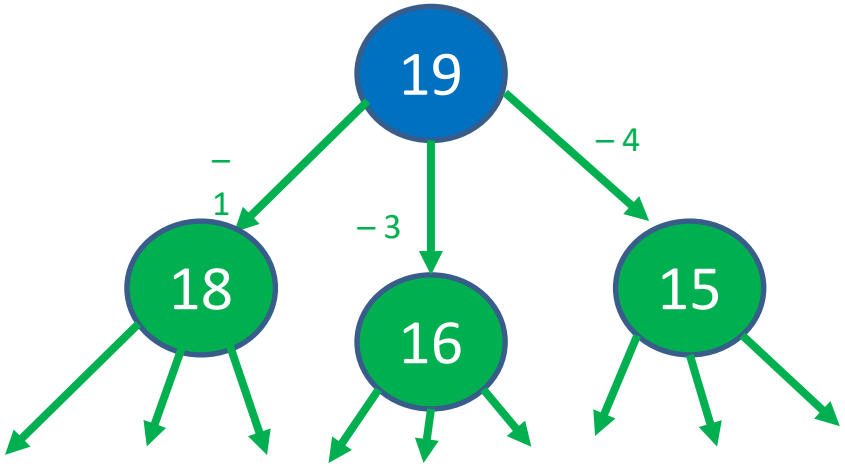
-- giving **five coins** in total.

For **18¢** ? (ans: three 4¢, and two 3¢ -- giving **five coins** in total)

An Exhaustive search approach to the minimum coins change problem

make **every possible combination of coins** to the change amount and choose the best (fewest).

Think of **problem space** here as a **graph** with **nodes** representing the **amount of change still to give** and directed edges showing possible reductions in that amount by issuing of a further **single coin**.

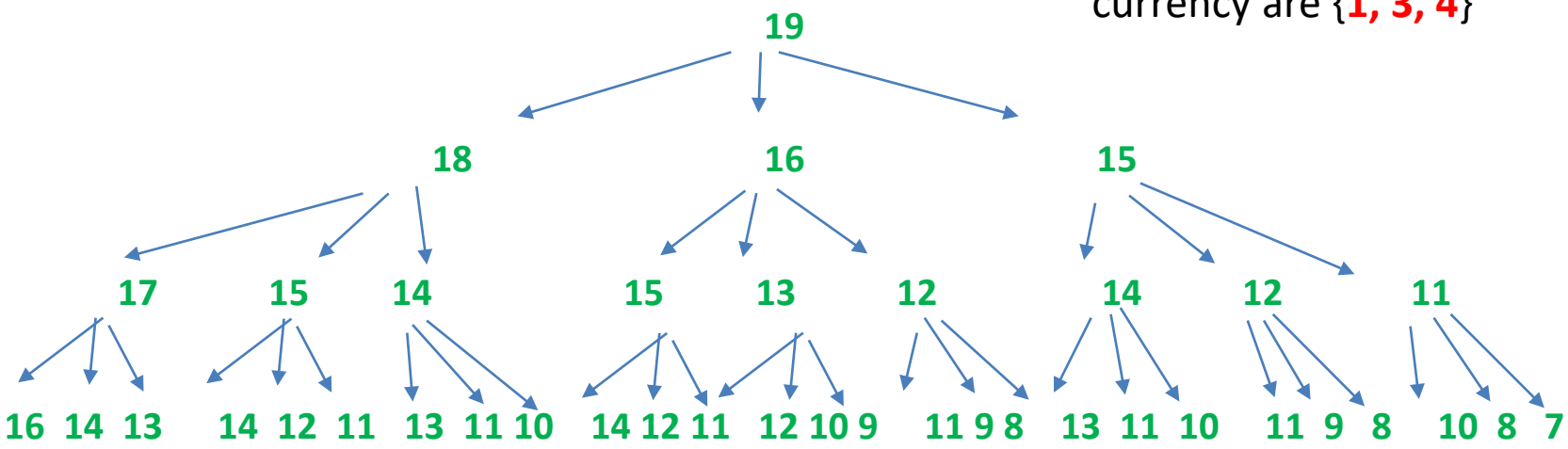


Eg. If denominations in our currency are {**1, 3, 4**}

Note: A directed graph helps visualise the algorithm.

An Exhaustive search approach to the change problem continued

Eg. If denominations in our currency are {**1, 3, 4**}



Level 4 (**81** nodes)

Level 5 (**243** nodes)

An exhaustive search approach may need to generate around **350 nodes** before it found the **minimum coin mix**!

More efficient algorithm approaches ?

Greedy strategy !

General idea: **don't try every state** in the problem space, at each point of decision or generation, **select** the most promising-looking option *at each decision point*.

So, for coin change, always **select** the **largest possible coin** denomination to issue next.

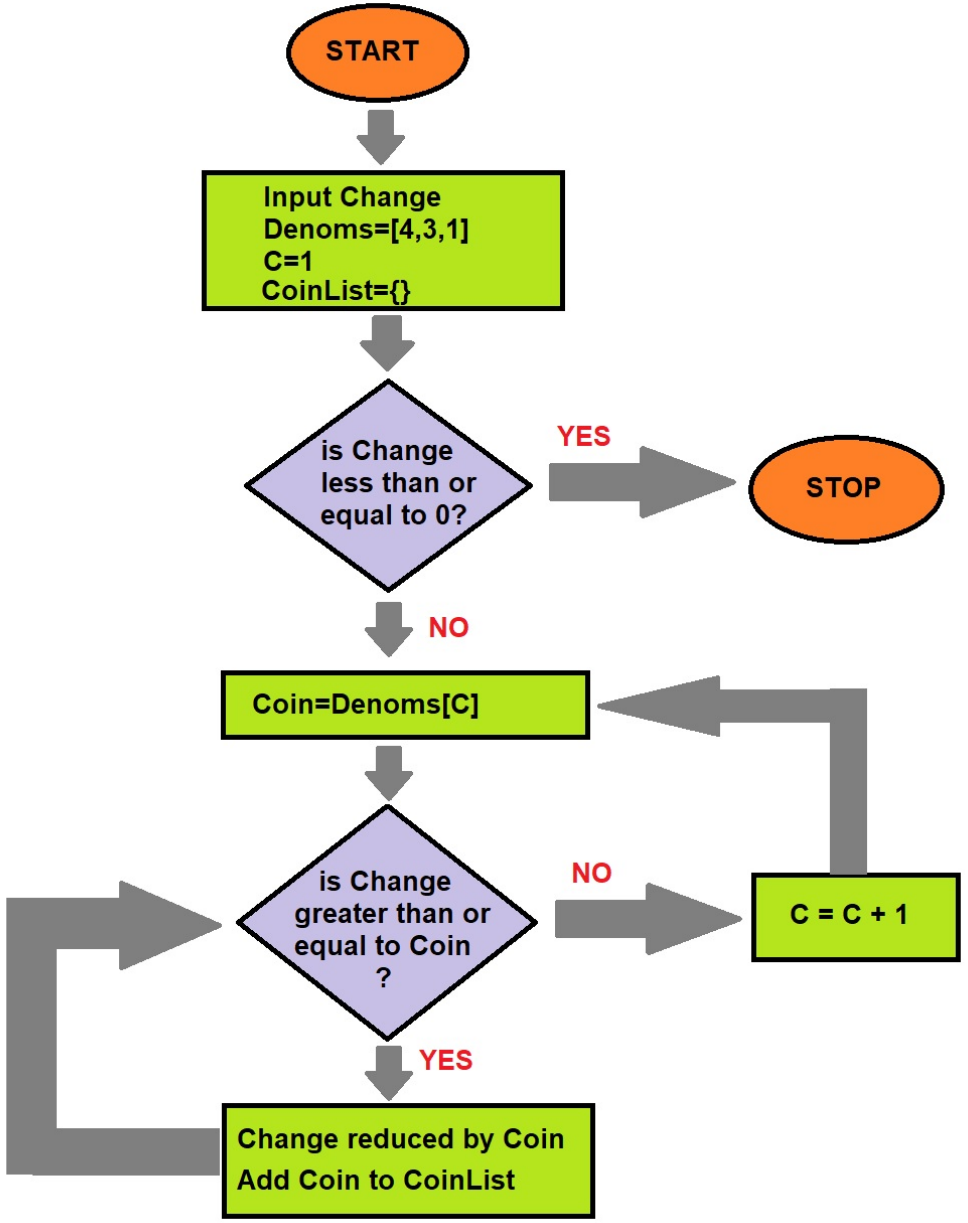
Greedy minimum coin change Algorithm in plain English with numbered steps:

1. **Choose** highest denomination coin where coin value ≤ remaining change

2. Use the denomination value found in **step 1.** as often as possible to **reduce remaining change**, adding that denomination to the **coin list** each time.

3. **If** remaining change is non-zero, **then** go to step 1. **else** output **coin list**.

Greedy minimum coin change Algorithm in a flowchart:



Note: A directed graph defines the algorithm.

Greedy minimum coin change Algorithm in structured pseudocode:

```
Algorithm MinimumCoinsGreedy (Change)
// Input: Change, the integer change required
// Output: CoinList, a list of the minimum coins required to make up change
AvailableDenominations := [4,3,1] //denominations array sorted in descending order
C := 1
Repeat until Change ≤ 0
    Coin := AvailableDenominations[C] //take next largest coin denomination from array
     While Change ≥ Coin do
        Change := Change – Coin
        Add Coin to CoinList
     End do
     C := C + 1
End Repeat
Return CoinList
End Algorithm
```

Stepping through the Greedy coin algorithm

Eg. If denominations in our currency are {**1, 3, 4**}

Let's check that out on Change = 19¢

19-4

15-4

11-4

7-4

3-4<0

so try **3-3** = 0

and Bingo! For 19¢ **Coinlist = <4, 4, 4, 4, 3>**

Only about **5 steps** or '**iterations**.' (Compared to around **350** by **exhaustive search!)**

But does this Greedy algorithm always find **the best possible solution?** **Let's check it on Change = 18¢**

Stepping through the Greedy coin algorithm

Eg. If denominations in our currency are {**1, 3, 4**}

Let's check that out on Change = 18¢

18-4

14-4

10-4

6-4

2-4<0

2-3<0

2-1

2-1

and Bingo! For 18¢ **Coinlist = <4, 4, 4, 4, 1, 1>**

Greedy algorithm does not find **the best possible solution for** 18¢ can be <4, 4, 4, 3, 3> **?**

Summary

Given a particular kind of problem, we start by **imagining** the **problem space** for that problem.

We imagine the **states** an algorithm will need to **step through** in order to get from the **inputs** to the **goal** or solution.

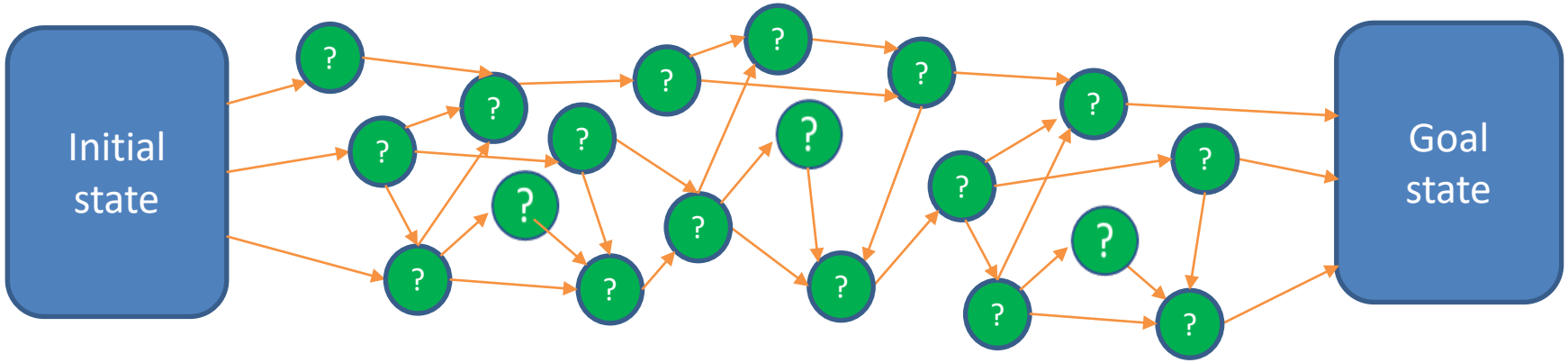
- **Exhaustive search**: systematically **create every possible state** in the problem space, **checking** each to see if it is a **solution** as you go.

  Can be time consuming!

- **Greedy**: **don't try every state** in the problem space, at each point of decision or generation, **select** the most promising-looking alternative *at each point*.

  Doesn't always find the **best solution**!

Graphs aren't just for information modelling. Defining algorithms with Graphs

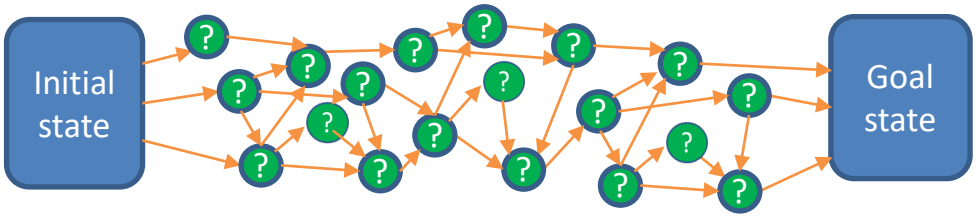


Defining algorithms with Graphs

A farmer returns from the market, where he bought a chicken, a bag of wheat and a dog. On the way home he must cross a river.

- **His boat is very small and won't fit more than one of his purchases.**
- **He cannot leave the chicken alone with the wheat (because the chicken would eat it).**
- **He cannot leave the chicken alone with the dog (because the chicken would be eaten).**

How can the farmer get everything on the other side in this river crossing puzzle?



D. DEFINING ALGORITHMS HOLIDAY HOMEWORK

Defining Algorithms
Holiday Homework

2024 Algorithmics Holiday Homework

- **Task 1** River Crossing problem – solution modelled using a **graph** (nodes, edges)
- **Task 2** Egyptian Fractions solved using the following methods
  a) **Exhaustive Search**
  b) **Greedy**

Task 1: River Crossing Problem

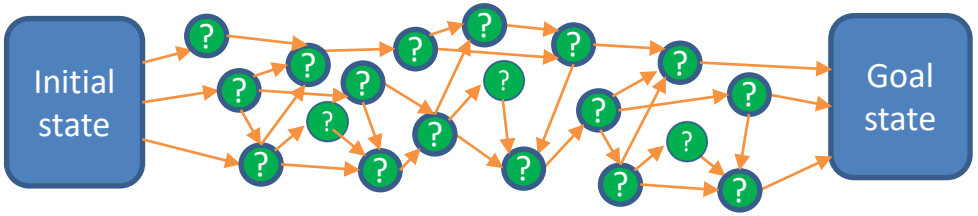
Defining algorithms with Graphs

A farmer returns from the market, where he bought a chicken, a bag of wheat and a dog. On the way home he must cross a river.

- **His boat is very small and won't fit more than one of his purchases.**
- **He cannot leave the chicken alone with the wheat (because the chicken would eat it).**
- **He cannot leave the chicken alone with the dog (because the chicken would be eaten).**

How can the farmer get everything on the other side in this river crossing puzzle?

Model the solution using a graph from initial to goal state.



Task 2:
Egyptian Fractions

Every positive proper fraction can be represented as **sum of unique unit fractions**.

A fraction is a unit fraction if the numerator is 1 and the denominator is a positive integer, for example 1/3 is a unit fraction.

Such a representation of a sum of unique unit fractions is called an Egyptian Fraction as it was used by ancient Egyptians.

Following are few examples:

- Egyptian Fraction Representation of **2/3** is **1/2 + 1/6**
- Egyptian Fraction Representation of **6/14** is **1/3 + 1/11 + 1/231**
- Egyptian Fraction Representation of **12/13** is **1/2 + 1/3 + 1/12 + 1/156**

Exhaustive Search Algorithm for finding Egyptian Fractions

For a given number of the form ‘X/Y’ where Y > X, first find the greatest possible unit fraction, then repeat for the remaining part.

Eg: consider 3/7, we first try 1/2 , but 3/7 < 1/2
We then try 1/3, 3/7-1/3 =9/21-7/21=2/21
We then try 1/4, but 2/21 < 1/4
We then try 1/5, but 2/21 < 1/5
……..
……..

Exhaustive Search tries out all the possible options.

Task 2 a:
Exhaustive Search Algorithm for determining Egyptian Fractions.

Describe in English
or
Define in a Flowchart
or
Define in Structured Pseudocode

Exhaustive Search Algorithm for converting a fraction $^{x}/_{y}$ into Egyptian Fractions

Algorithm EgyptianBruteForce (x,y)

// Input x, the integer numerator of the fraction

// Input y, the integer denominator of the fraction

// Assumption x < y

// Output a list of Egyptian Fractions that add to x/y

Exhaustive search tries out all the possible options.

Greedy Algorithm for Egyptian Fractions

For a given number of the form 'X/Y' where Y > X, first find the greatest possible unit fraction, then repeat for the remaining part.

Eg: consider $\frac{3}{7}$, find the nearest unit fraction less than $\frac{3}{7}$, think about how many times 3 goes into 7, then always round up, the ceiling function always rounds up.

ceiling($\frac{7}{3}$)=3, so try 1/3, $\frac{3}{7} - \frac{1}{3} = \frac{9}{21} - \frac{7}{21} = \frac{2}{21}$

ceiling($\frac{21}{2}$)=11, so try 1/11, $\frac{2}{21} - \frac{1}{11} = \frac{22}{231} - \frac{21}{231} = \frac{1}{231}$

Note: ceiling is a function that always rounds up eg. ceiling($\frac{3}{7}$) will return 1.

Greedy tries out the next best possible option.

Task 2 b:
Greedy Algorithm for determining Egyptian Fractions.

Describe in English
or
Define in a Flowchart
or
Define in Structured Pseudocode

A Greedy Algorithm for converting a fraction $^{x}/_{y}$ into Egyptian Fractions

Algorithm EgyptianGreedy (x, y)

// Input x, the integer numerator of the fraction

// Input y, the integer denominator of the fraction

// Assumption x < y

// Output a list of Egyptian Fractions that add to x/y

RMIT VXLabs Wed 6th Dec

- Algorithmics Excursion
- Leaving from State Library Victoria (SLV) at 9.15am, please be on time
- Building 91, 110 Victoria St Carlton
- Walking distance from SLV