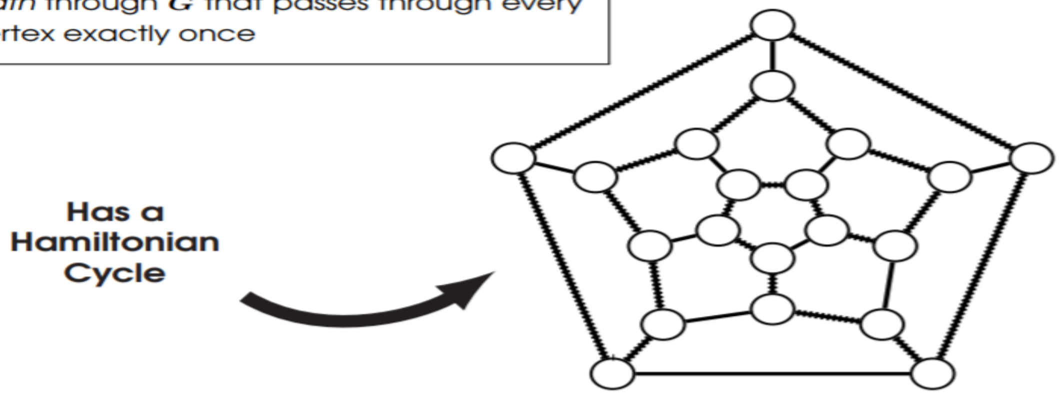
The Hamiltonian Cycle Problem

- That is there is a path that starts end ends at the same nodes, and it only visit one node once.
- Naively: Try every possibility, brute-force it. 20! Possible combination
- Non-deterministic Turing Machine can solve this quickly because each cycle can be split between the processors, so all 20! can run in parallel, which will be in polynomial time to check weather is a Hamiltonian cycle. Therefore, this is a problem that belongs to the class NP.

Input) An undirected graph $G = (V, E)$

Output) **True** if and only if there is a *closed path* through G that passes through every vertex exactly once

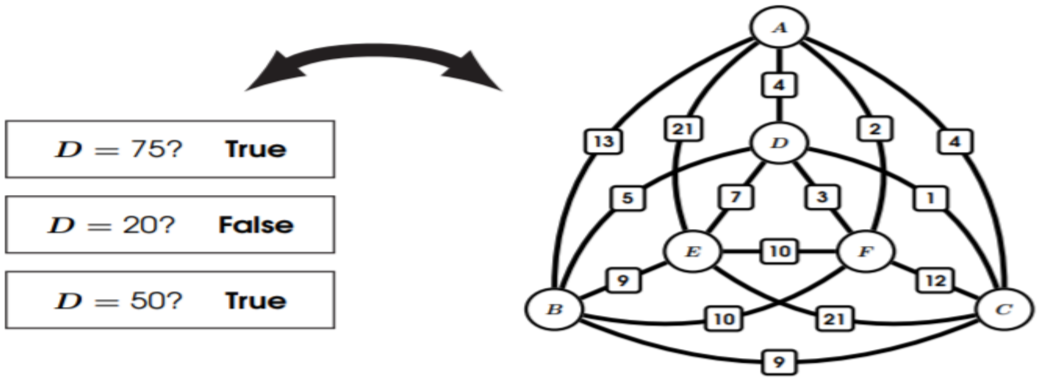


The Travelling Salesman Problem

- Given value **D** we must evaluate if it is possible to make on the graph.
- Non-deterministic Turing Machine can solve this problem quickly because it can split checking the paths in parallel on a different processor. Therefore, that's why it is NP.

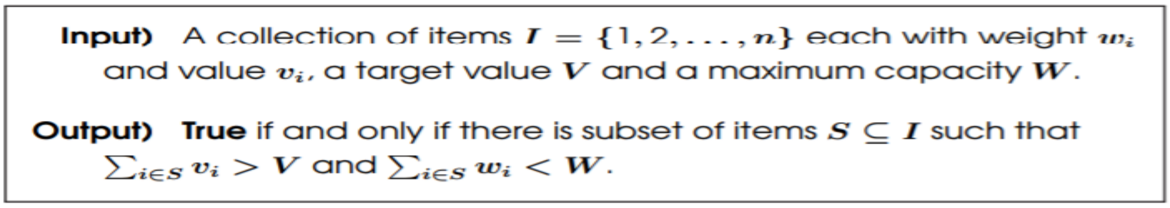
Input) A complete weighted graph $G = (V, d)$, where $d : V \times V \to \mathbb{R}$, and target distance $D > 0$.

Output) **True** if and only if there is a *closed path* through G of length $< D$ that passes through every vertex exactly once



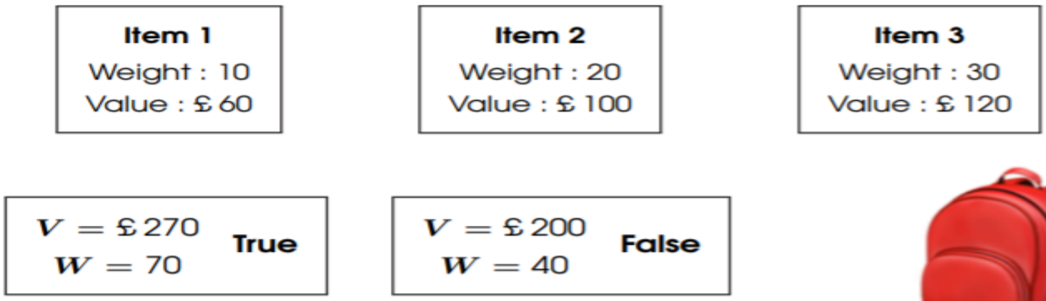
The Knapsack Problem

- So, suppose you are robbing into apple store, with limited capacity in the bag, this problem can evaluate to true, if there is a subset of items that are worth highest price and within the possible capacity.



Input) A collection of items $I = \{1, 2, \ldots, n\}$ each with weight w_i and value v_i, a target value V and a maximum capacity W.

Output) **True** if and only if there is subset of items $S \subseteq I$ such that $\sum_{i \in S} v_i > V$ and $\sum_{i \in S} w_i < W$.

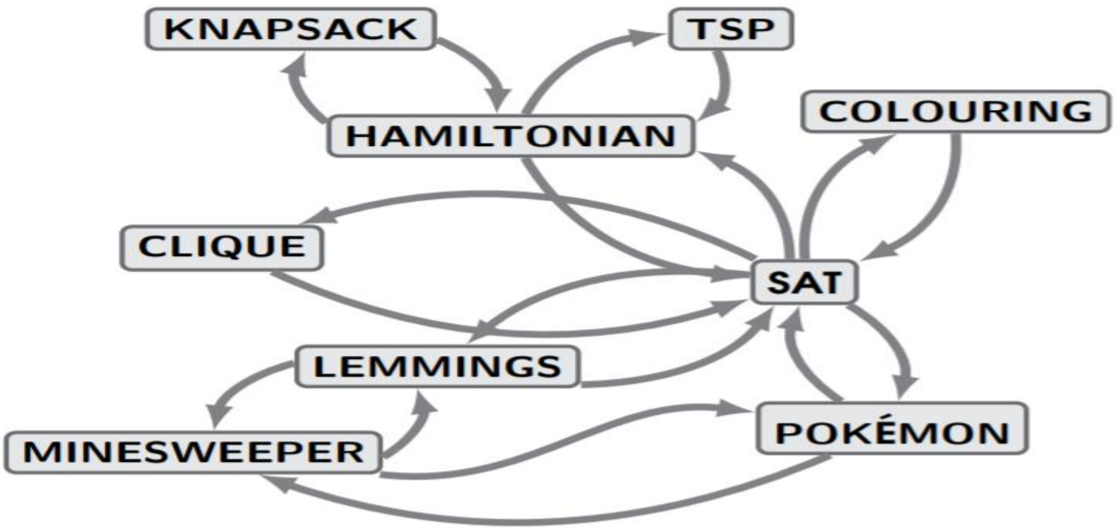


Item 1	Item 2	Item 3
Weight : 10	Weight : 20	Weight : 30
Value : £ 60	Value : £ 100	Value : £ 120

$V = £\,270$, $W = 70$ **True**

$V = £\,200$, $W = 40$ **False**

Relationships between NP-complete Problems



The Cook-Levin Theorem

(The Existence of NP-Complete Problems)

Theorem (The Cook-Levin Theorem) The Boolean Satisfiability problem **SAT** is **NP**-complete.

(*i.e.* SAT is among the hardest problems in NP)

Proof (sketch): **(you do NOT need to memorise this proof)**

We want to show that **every** problem $X \in$ **NP** can be reduced to **SAT**.

Step 1) Let $X \in$ **NP** be **any problem** belonging to the class **NP**.

By **definition** there is some NDTM $\mathcal{M}$ such that

$\mathcal{M}$ has a polynomial-length computation that accepts w $\iff$ $w \in X$

(for all input words $w \in \Sigma^*$)

Step 2) For each **state** $q \in Q$, **symbol** $a \in \Sigma$ and **integers** $i, t \leq T(n)$

$\mathbf{cell}_{i,t}(a)$	$=$	Cell i of the $\mathcal{M}$´s tape contains symbol a at time t
$\mathbf{head}_{i,t}$	$=$	The tape head is in position i at time t
$\mathbf{state}_{q,t}$	$=$	The machine is in state q at time t

(where $T(n) \in O(n^k)$ is the worst-case termination-time of $\mathcal{M}$)

Step 3) With these **propositional variables**, we make the following claim:

Claim: We can write down a propositional formula that describes the behaviour of a NDTM on a given input

$F_{\mathcal{M},w}$ $=$ $\mathcal{M}$ has a computation that accepts w

Step 4) We then have that

$w \in X$ $\iff$ $F_{\mathcal{M},w}$ if satisfiable

which is to say that $X \leq_p$ **SAT**.

Step 5) Since we have assumed **nothing special** about X other than it can be solved by a NDTM in polynomial time (and space), we must have that

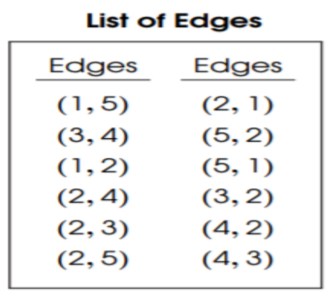
$X \leq_p$ **SAT** for all $X \in$ **NP**

which is to say that **SAT** is **NP-complete**

Q.E.D.

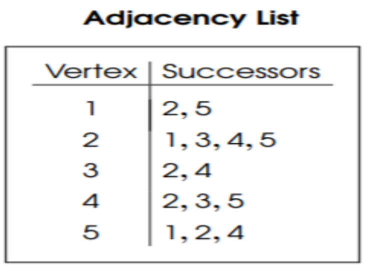
Graph Algorithms

- Data Structures for Graphs
    - The type of **data structure** used to store a graph can affect the efficiency of your algorithms, and memory requirements.



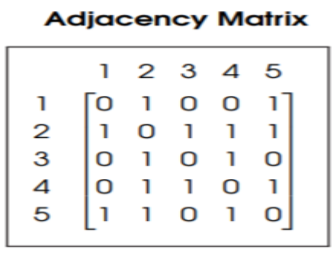
List of Edges

Edges	Edges
(1, 5)	(2, 1)
(3, 4)	(5, 2)
(1, 2)	(5, 1)
(2, 4)	(3, 2)
(2, 3)	(4, 2)
(2, 5)	(4, 3)



Adjacency List

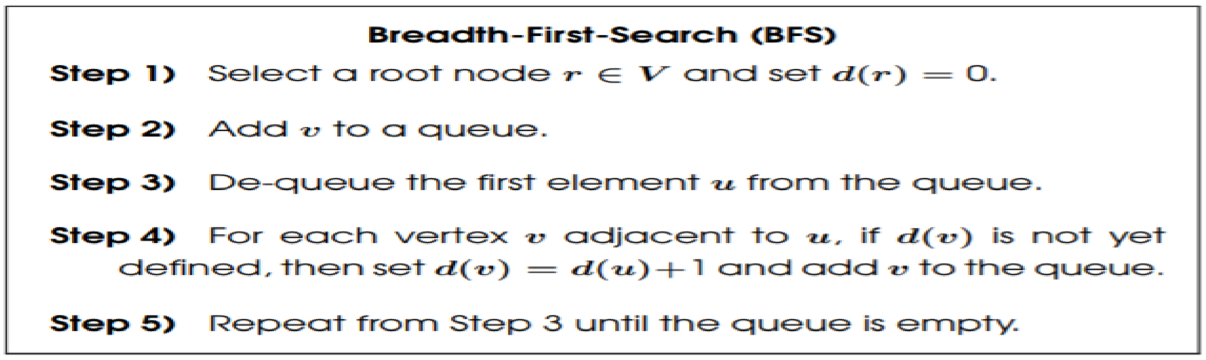
Vertex	Successors
1	2, 5
2	1, 3, 4, 5
3	2, 4
4	2, 3, 5
5	1, 2, 4



Adjacency Matrix

$$\begin{array}{c|ccccc} & 1 & 2 & 3 & 4 & 5 \\ \hline 1 & 0 & 1 & 0 & 0 & 1 \\ 2 & 1 & 0 & 1 & 1 & 1 \\ 3 & 0 & 1 & 0 & 1 & 0 \\ 4 & 0 & 1 & 1 & 0 & 1 \\ 5 & 1 & 1 & 0 & 1 & 0 \end{array}$$

Breadth-First-Search



Breadth-First-Search (BFS)

Step 1) Select a root node $r \in V$ and set $d(r) = 0$.

Step 2) Add v to a queue.

Step 3) De-queue the first element u from the queue.

Step 4) For each vertex v adjacent to u, if $d(v)$ is not yet defined, then set $d(v) = d(u) + 1$ and add v to the queue.

Step 5) Repeat from Step 3 until the queue is empty.

- Step 1) Pick a root note when search starts from
- Step 2) Add all vertices that can be accessed from to front node to the queue
- Step 3) remove element that is in front of queue
- Step 4) add all nodes to queue that was accessible from the removed node, no repetition
- Step 5) Repeat from Step 3 until the queue is empty.

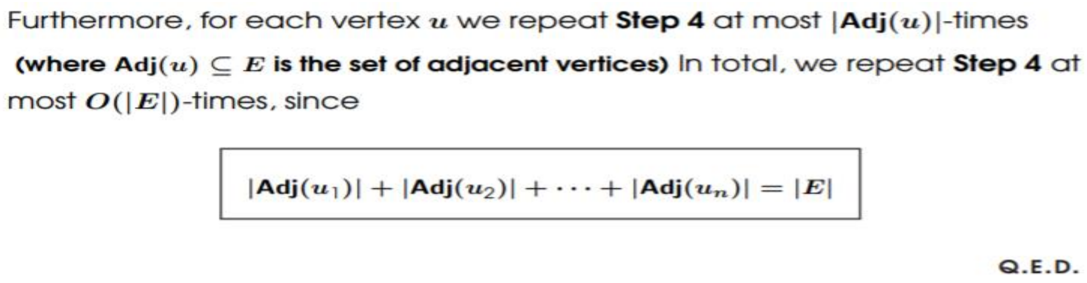
The point is to measure the distance of all nodes from the root in some order.

If we don't check If the node is already in the queue, we could end up in an infinite loop.

***Theorem**: The worst-case termination time for Breadth-First-Search is **O(|v| + |e|)** which is Big oh of cardinality of vertices plus cardinality of the edges. Takes polynomial time.*

***Proof**:*

- Each vertex is enqueued at most once and so the number of times that **Step 3** and **Step 5** are repeated is at most **O(|v|)** times.

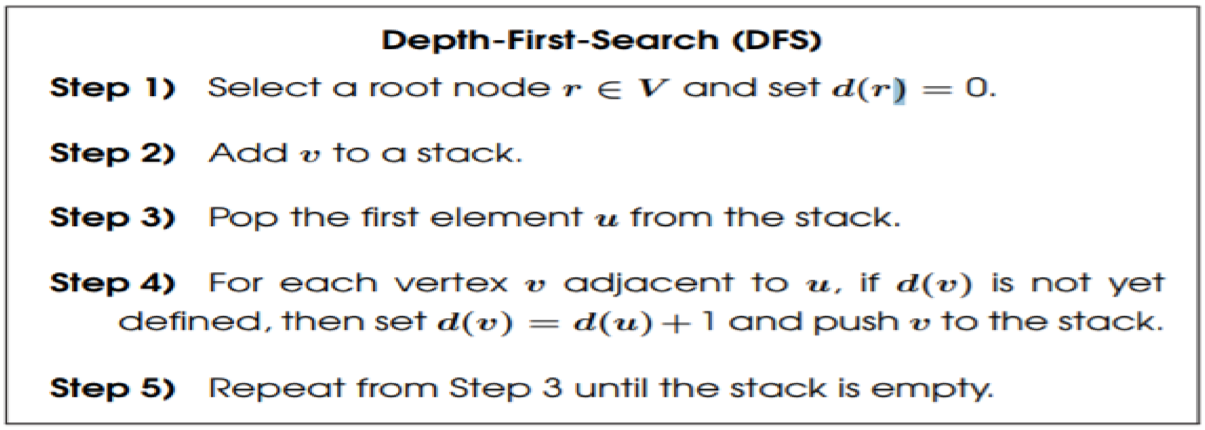


Furthermore, for each vertex u we repeat **Step 4** at most |**Adj**(u)|-times **(where Adj$(u) \subseteq E$ is the set of adjacent vertices)** In total, we repeat **Step 4** at most $O(|E|)$-times, since

$$\boxed{|\mathbf{Adj}(u_1)| + |\mathbf{Adj}(u_2)| + \cdots + |\mathbf{Adj}(u_n)| = |E|}$$

Q.E.D.

Depth-First-Search



Depth-First-Search (DFS)

Step 1) Select a root node $r \in V$ and set $d(r) = 0$.

Step 2) Add v to a stack.

Step 3) Pop the first element u from the stack.

Step 4) For each vertex v adjacent to u, if $d(v)$ is not yet defined, then set $d(v) = d(u) + 1$ and push v to the stack.

Step 5) Repeat from Step 3 until the stack is empty.

Theorem: The worst-case termination time for Depth-First-Search is **O(|V|+|E|)**

Proof: The poof is almost identical to that of the **Breadth-First-Search,** replacing the **queue** for a **stack.**