(Sec. 3.3 pp. 182-187, & 7.4 p299). **Algorithms, Graphs, SAT**

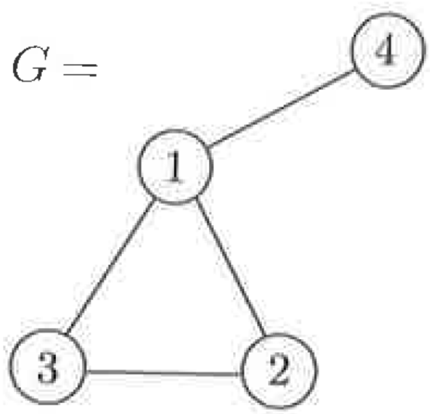
- Key distinction re TMs and languages
  - TM T **recognizes** L if for all w in L T accepts w
    - Says nothing about what if w not in L
  - TM **decides** L if
    - T recognizes L
    - If w not in L, T always halts (in reject state)
- Hilbert's 10^{th} problem (1900): *Can any algorithm tell if a polynomial equation has any integer roots?*
  - Sample polynomial equation: $6x^3yz^2+3xy^2-x^3-10=0$
  - Example does at x=5, y=3, z=0
  - Critical point: we want **yes/no** answer for any polynomial
  - 1970: no such algorithm exists
- Key starting point: what is an "algorithm"?
- Key Definition: 1936 **Church-Turing Thesis**
  - Any function over the natural #s is computable by a algorithm iff it is computable by a TM
  - Each transition of a TM is a "**step**"
    - Step takes finite time
  - Finite # of steps to get to accepting state
- "*Does algorithm exist*" eqvt to "*Is there a TM decider*"

- Back to Hilbert
  - Define D = {p|p is a polynomial with an integral root}
  - D is **recognizable**:
    - Consider D_1={p|p a polynomial over single variable x with an integral root}
    - Recognizing TM M_1: Assume input string defines a p
      - Start an enumerator to generate 0, 1 -1, 2, -2, …
      - For each value compute p at that value
      - If a root, halt and accept
    - Note: if p has no integral roots, M_1 loops
    - TM recognizer for general D generates all cases of integers 1 at a time
  - Hilbert's 10^{th} problem equivalent: does some TM **decide** D
    - I.e. Does some TM ***always halt*** for any p
  - For D_1 (exactly 1 variable) there are bounds that can constrain solution space (see p. 184 and problem 3.21)
    - Thus we can halt M_1 as soon as we reach these bounds
    - Thus modified M_1 is a **decider** for D_1
  - Theorem from 1970: no such bounds exist for multi-variable polynomials
    - **Cannot construct a decider for D** same way as for D_1
- When deciders exist: ***do polynomial time TMs exist?***

- (p. 184) Terminology for describing TMs
  - (p. 185) 3 ways for describing TMs
    - **Formal Description:** 7 tuple and δ
    - **Implementation Description**: use English prose to describe tape movements and tape writing
    - **High-level Description**: English prose to describe algorithm, ignoring implementation details
      - Often building one TM out of composition of others
  - (p.185)Notation for describing TM tapes(esp. initial tapes)
    - Tape always contains a **string**
    - Use strings to represent objects (#s,grammars, graphs..)
    - TM can be written to "decode" string representations
    - Notation for string representation of object O**: <O>**
    - Notation for multiple objects $O_1, O_2, \ldots O_k = \langle O_1, O_2, \ldots O_k \rangle$
    - TM algorithm described as indented lines of text
      - Each a **stage**: multiple TM operations
      - Assume initial stage checks format of input tape

- (p 186) **Graphs**
  - set of **vertices**, each encoded as different positive #
    - Note: book calls vertices as **nodes**
  - set of **edges** between vertices, each encoded as tuple of 2 vertices
    - edges may be **directed** (from to) or **undirected**
      - Undirected edge eqvt to pair of directed edges
  - Example of undirected graph

$G =$



$\langle G \rangle =$

(1,2,3,4)((1,2),(2,3),(3,1),(1,4))

- A graph is **connected** iff every vertex can be reached from every other vertex by some path of edges

- (p. 186) A = {<G>| G is a connected undirected graph}
  - <G> = string of symbols representing two lists:
    - "(" list of vertex #s separated by "," ")"
    - "(" list of edges separated by "," ")"
      - Each edge: "(" <vertex 1> ",", <vertex 2> ")"
- A TM decider algorithm for testing connectedness:

M = "On input <G>, the encoding of graph G:

0. Verify <G> is formatted properly & reject if not
1. Select 1st vertex of G and "**mark**" it
   - "Marking" adds a ***** ("dot") to leftmost symbol
2. Repeat until no new vertices unmarked: For each vertex in G, mark it if it is attached by an edge to a vertex that is already marked
   1. Scan vertex list to find an unmarked vertex n_1
      - **Underline** 1st symbol
   2. Scan vertex again and find 1st dotted vertex n_2
      - Underline that also
   3. For each edge in edge list see if (n_1, n_2) or (n_2, n_1): If so
      - Dot the undotted vertex; Remove both underlines
      - Restart major step 2
3. Scan all vertices of G to determine if all are "marked"
   - If yes, accept; if no reject

- Clearly this always halts on valid <G>: only finitely many vertices to scan
- Also clearly polynomial time algorithm
- Equivalent to **Breadth First Search** Algorithm (**BFS**)
  - Basis for the **GRAPH500** benchmark
    - www.graph500.org
    - Literally thousands of different implementations on different computers, esp. parallel
    - Established by an **ND quad-domer**
- Many other important Graph Algorithms
  - Shortest path between 2 vertices
    - BFS with a count of # of edges
  - Are some vertices in a “cycle”
    - Variation of BFS
  - Traveling Salesman problem
    - Much, much harder
  - See https://en.wikipedia.org/wiki/Category:Graph_algorithms

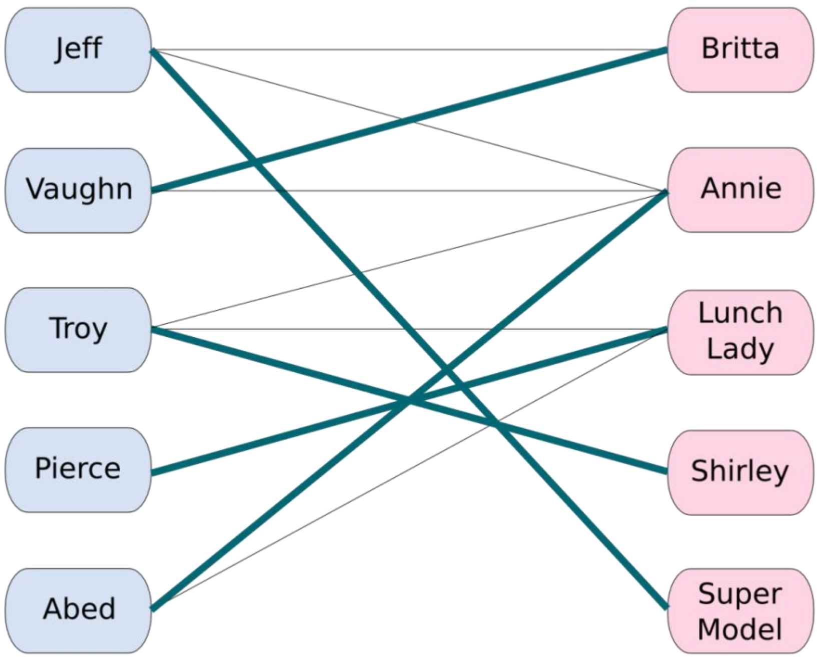
- (p. 299) **SAT: Boolean Satisfiability**
- **SAT** ={<wff>|wff a satisfiable Boolean formula}
  - **wff** is well-formed-formula constructed from
    - V Boolean variables
    - Boolean operations AND, OR, NOT
  - **Satisfiability**: is there a substitution of 0s and 1s to variables that makes the wff true
    - i.e. makes all clauses simultaneously true
  - **Unsatisfiability** if no substitution makes all clauses true at same time
  - See: https://en.wikipedia.org/wiki/Boolean_satisfiability_problem
- **Clausal form**:
  - **wff** restructured as AND of a set of clauses
  - Each **clause** an OR of a set of literals
  - Each **literal** a variable or its negation
- For a wff in clausal form to be true
  - *All* clauses must be true
  - For any clause to be true at least one literal must be true
- Clearly there is a polynomial time verifier
  - Given list of variables and their values
  - Scan each clause, looking up value for each literal

- What is easiest approach to decidability?
  - Build truth table with a row for each possible assignment
  - But for V variables there are 2^V rows, so this is **exponential**!
  - Can we ever do better?
- **1SAT** is trivially polynomial (linear)
  - Each clause is one literal
  - If any 2 clauses are a variable & its complement, then reject
- What about **2SAT**?
  - Each clause has exactly 2 literals
    - $C_i = (L_{i1} \vee L_{i2})$, L_{i1}, L_{i2} are literals from different variables
  - (x v y) can also be written as ~x => y, or as ~y => x
    - If x is false then <u>y must be true</u>
    - And if y is false then <u>x must be true</u>
- Create a graph from the wff
  - 1 vertex for each possible literal
    - eqvt to 2 vertices for each variable
      - i.e. 1 for a variable, and 1 for its negation
  - For each clause, create 2 edges following the implications

- Now if some variable has an assignment
  - Start with the vertex for the matching literal which is now false
  - Follow all paths from that vertex (the BFS algorithm)
    - This is all the literals which now must be true
  - If you ever get the negation of the original literal, then a contradiction, AND NO ASSIGNMENT IS POSSIBLE
    - Equivalent to finding a **cycle** in the graph
- But we know that BFS is polynomial
  - And we need only apply the test for each of V variable
- **So 2SAT is also polynomial**
- Example: $(\neg x \vee y) \wedge (x \vee y) \wedge (x \vee \neg y) \wedge (\neg x \vee \neg y)$
  - 4 Clauses, 2 variables, 4 literals
  - 4 vertices: x, y, $\neg x$, $\neg y$
  - 8 matching edges:
    - (x,y), ($\neg y$, $\neg x$)
    - ($\neg x$,y), ($\neg y$,x)
    - ($\neg x$, $\neg y$) , (y, x)
    - (x, $\neg y$), (y, $\neg x$)
  - Path from $\neg x$ to y to x, so this is unsatisfiable

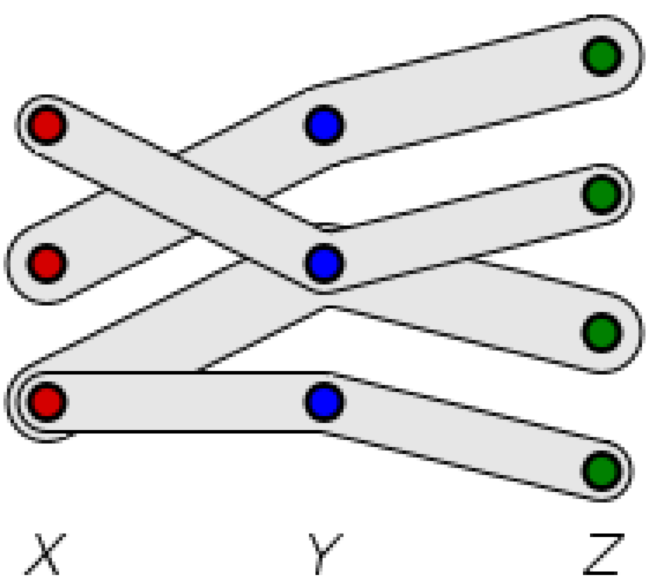
- What about **3SAT** and above?
  - 3SAT: all clauses have 3 literals (L_1, L_2, L_3)
  - All bigger SAT problems can be converted into 3SAT
  - So decidability of general SAT eqvt. to decidability of 3SAT
- Many real problems have millions of variables
  - Truth Table of $2^{|V|}$ thus monstrous
- Key result: No **known polynomial time decider algorithm**
  - Virtually all include some sort of "**guess and backtrack**"
- Further: Large class of other problems can be shown eqvt. to SAT
- Thus there is a large class of real-world problems for which no polynomial-time TM appears to exist

- **Bipartite Matching Problem** (aka **Marriage Problem**)
  - Given 2 sets $A = \{a_1, \ldots a_{|A|}\}$ & $B = \{b_1, \ldots b_{|B|}\}$ of vertices
  - and set E of edges e_{ij} between a_i to b_i
  - Is there a subset of edges where every vertex has at most 1 edge?



- **Perfect Matching**: is there a matching which includes all vertices
  - Known best algorithms $O(|V|^{2.4})$ or $O(|E|^{10/7})$
- **Maximal Matching**: what matching maximizes the number of vertices involved (not a decision problem)

- E.g. Bipartite Matching converts to a 2SAT problem
  - Variables: one x_{ij} for each edge e_{ij}
    - Assigning a 1 says a_i and b_j are matched by this edge
    - Assigning a 0 says they are NOT matched by this edge
  - For each vertex a_i, generate a set of clauses ($\sim x_{ij}$, $\sim x_{ik}$) for all j's and k's for which edges from vertex a_i exist
    - This prevents multiple edges from being selected from a_i at same time
  - If variables for any 2 edges were true, then some clause is false.
    - Large # of vertices but still polynomial
- What about "Tripartite" and above? – same as 3SAT
  - **No known polynomial decider algorithms**



https://upload.wikimedia.org/wikipedia/commons/thumb/5/50/3-dimensional-matching.svg/240px-3-dimensional-matching.svg.png