COT 6405
ANLYSIS OF ALGORITHMS

NP - Completeness

Computer & Electrical Engineering and Computer Science Dept.
Florida Atlantic University

Spring 2017

Outline

- Polynomial time
- Polynomial-time verification
- NP-completeness and reducibility
- NP-completeness proofs
- NP-complete problems

Reference: CLRS ch 34

Overview

- Polynomial-time algorithms: for input size n, their worst-case RT = $O(n^k)$, for some const k
- Some problems can be solved with larger complexity
  $\Rightarrow$superpolynomial time
- Others cannot be solved no matter how much time is allowed
  - Turing's Halting problem
- Problems solvable in:
  - polynomial time, are ***tractable*** or easy
  - superpolynomial time, are ***intractable*** or hard

NP-Complete problems

NP-Complete problems

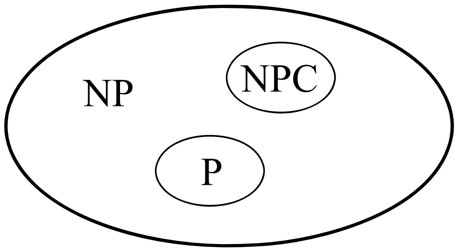
- Nondeterministic polynomial time (NP) complete problems
- Status is unknown:
  - no polynomial-time algorithm discovered for any NP-complete problem
  - no formal proof that such an algorithm is impossible
- 3 classes of problems: P, NP, NPC
  - P is polynomial time
  - NP is nondeterministic polynomial
  - NPC is NP-Complete

NP class

- problems "*verifiable*" in polynomial time
  - if we are given a "*certificate*" of a solution, we can verify that the certificate is correct in time polynomial in the size of the input problem
  - example: Hamiltonian-Cycle problem
- Any problem in P is also in NP
  - for now it is believed that $P \subseteq NP$
  - open question: $P \subset NP$?
  - open question since 1971: $P \neq NP$?

NPC class

- a problem is NPC if:
  - it is in NP
  - it is as “hard” as any other problem in NP
- if you prove a problem is NPC, then
  - you prove its ***intractability***
  - approaches:
    - design approximation algorithms
    - solve a tractable, special case
- how most theoretical computer scientists view relationship among P,NP,NPC



Optimization problems vs. decision problems

- Optimization problems
  - solutions have associated a value
  - e.g. finding shortest-path between u and v in an undirected graph G
- Decision problems
  - the answer is simply YES or NO (or “1” or “0”)
  - we usually can cast an optimization problem to a related decision problem by imposing a bound on the value to be optimized
  - e.g. given a directed graph G, vertices u and v, and an integer k, does it exist a path from u to v consisting of at most k edges?
- Theory of NPC works on decision problems
- If a decision problem is hard, the corresponding optimization problem is hard as well

A formal language framework

- alphabet $\Sigma = \{0,1\}$
- set $\Sigma^*=\{\varepsilon,0,1,00,11,01,10,000,\ldots\}$ – is the set of all binary strings
- a language L is a subset of Σ^*
- a decision problem Q can be represented as a language:

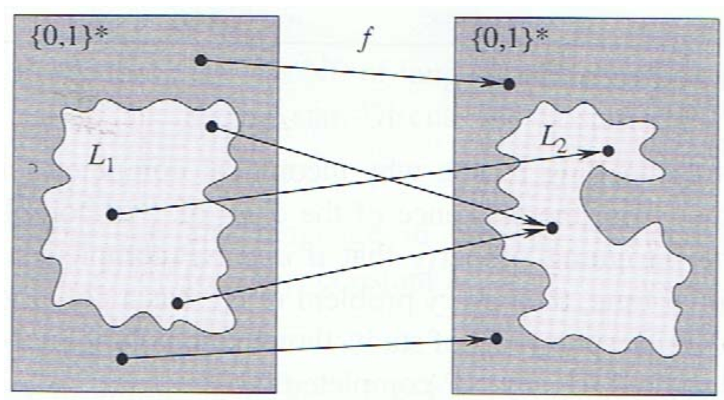
  $L = \{x \in \Sigma^*: Q(x) = 1\}$
- Example: decision problem PATH has the corresponding language:

  PATH = {<G,u,v,k>: G = (V,E) is an undirected graph, $u,v \in V$, $k \geq 0$ is an integer, and there is a path from u to v in G consisting of at most k edges}

NP-completeness and reducibility

- Reducibility *definition*:

L_1 is ***polynomial-time reducible*** to L_2, written $L_1 \leq_p L_2$ if there exists a polynomial-time computable function $f : \{0,1\}^* \rightarrow \{0,1\}^*$ s.t. for all $x \in \{0,1\}^*$, $x \in L_1$ iff $f(x) \in L_2$.



- f is called the reduction function
- L_1 is not more than a polynomial factor harder than L_2

NP-completeness and reducibility

- *Definition*:

A language $L \subseteq \{0,1\}^*$ is ***NP-complete*** if
1. $L \in NP$
2. NP-hard: $L' \leq_p L$ for every $L' \in NP$

- *Theorem*:

If an NP-Complete language L is polynomial-time solvable, then P = NP.

Proof:

$L \in NPC$, then for any $L' \in NP$, $L' \leq_p L \Rightarrow L' \in P$

$\Rightarrow NP \subseteq P \Rightarrow NP = P$

NP-completeness and reducibility

- *Theorem*:

Given a language L, if $L' \leq_p L$ for some $L' \in$ NP-complete, then L is NP-hard.

Proof:

$$\left.\begin{array}{l} L' \in \text{NP-complete} \Rightarrow L'' \leq_p L' \text{ for any } L'' \in NP \\ \qquad\qquad\qquad\qquad\qquad L' \leq_p L \end{array}\right\} \Rightarrow$$

$\Rightarrow L'' \leq_p L$ for any $L'' \in NP \Rightarrow L$ is NP-hard

A language $L \subseteq \{0,1\}^*$ is ***NP-complete*** if
1. $L \in NP$
2. NP-hard: $L' \leq_p L$ for some $L' \in$ NP-Complete

NP-completeness proof

Method to prove that a language L is NP-complete:

1. prove $L \in NP$
2. select a known NP-complete language L'
3. describe an algorithm (called *reduction algorithm*) that computes a function f mapping every instance x of L' to an instance f(x) of L
4. prove that function f satisfies:

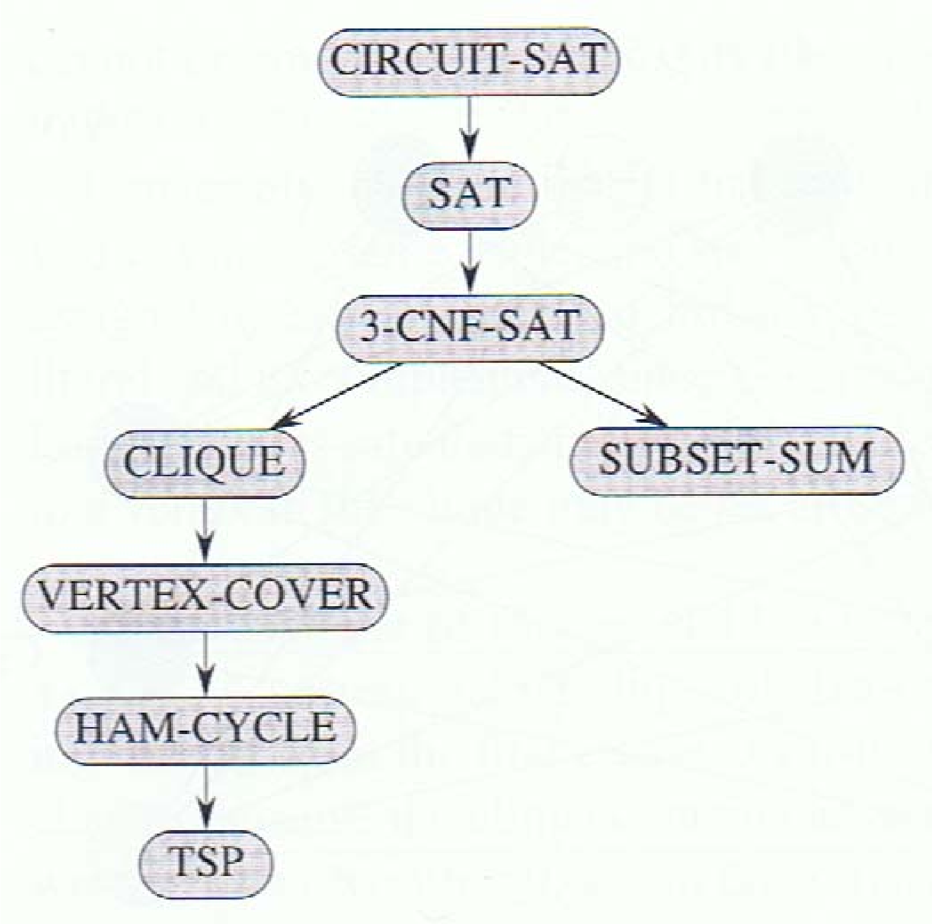
   $$x \in L' \text{ iff } f(x) \in L \text{ for all } x \in \Sigma^*$$

5. prove that the reduction algorithm runs in polynomial time

NP-completeness

- class P was introduced in 1964 by Cobham and independently in 1965 by Edmonds, who also:
  - introduced the class NP
  - conjectured that P $\neq$ NP
- 1971: notion of NP-completeness was proposed by Cook
  - he also showed that SAT and 3-SAT are NP-complete
- 1972: Karp introduced the reduction methodology
  - showed many problems are NP-complete: clique, vertex-cover (VC), hamiltonian-cycle (HC)
- … since then thousands of problems have been proven to be NP-complete

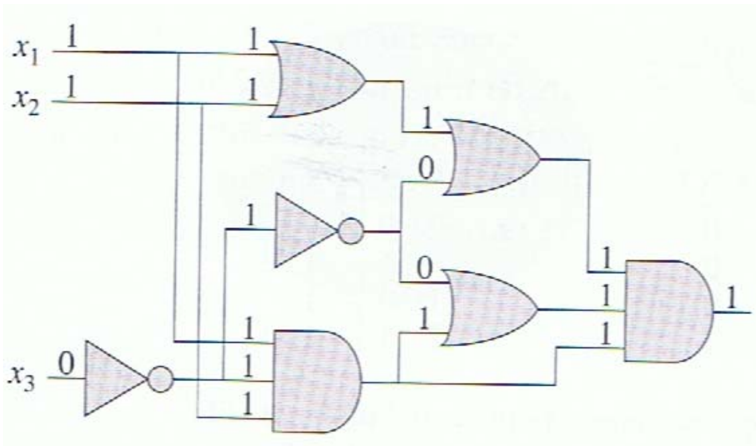
NP-complete problems



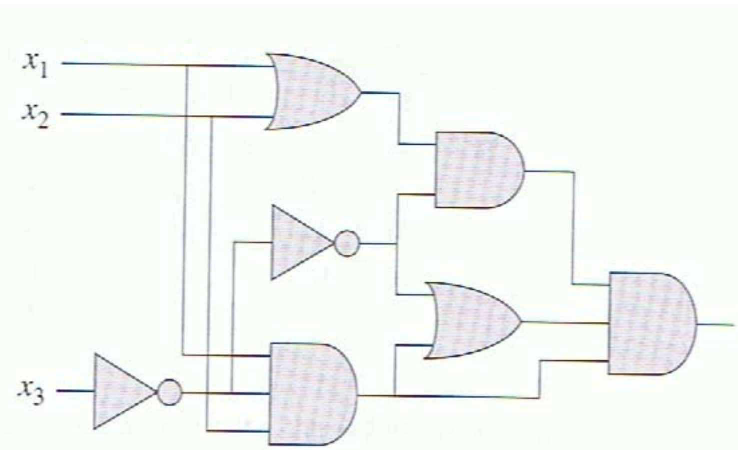
Circuit-Satisfiability (CIRCUIT-SAT)

CIRCUIT-SAT: given a combinatorial-circuit composed of AND, OR, and NOT gates, is it satisfiable?

- a circuit is *satisfiable* if there is an assignment for which the output is 1



circuit is satisfiable



circuit is unsatisfiable

CIRCUIT-SAT

- for n inputs, checking all possible inputs take $\Omega(2^n)$
  - superpolynomial
- Theorem: The circuit-satisfiability problem is NP-complete.

SAT (boolean formula satisfiability)

A ***boolean formula*** ϕ is composed of:

- n boolean variables $x_1, x_2, \ldots, x_n$
- m boolean connectives: $\wedge$ (AND), $\vee$ (OR), $\neg$(NOT), $\rightarrow$ (implication), $\leftrightarrow$ (if and only if)
- parentheses

A ***satisfying assignment*** is a truth assignment that causes it to evaluate to 1.

Example: $\phi = (x_1 \wedge x_2) \vee (\neg x_1 \wedge x_3)$

satisfying assignment: $< x_1=0, x_2=0, x_3=1 >$

SAT problem: given a boolean formula, is it satisfiable?

SAT

Solution in superpolynomial time:

- check every assignment for a satisfying assignment
- 2^n possible assignments $\Rightarrow$ RT = $\Omega(2^n)$

Theorem : SAT is NP-complete.

proof:

- SAT $\in$ NP

- a certificate with a satisfying assignment can be verified in polynomial time
- verifying algorithm:
  - replace each variable with the corresponding value
  - evaluate expression
  - if it evaluates to 1, then the formula is satisfiable

SAT

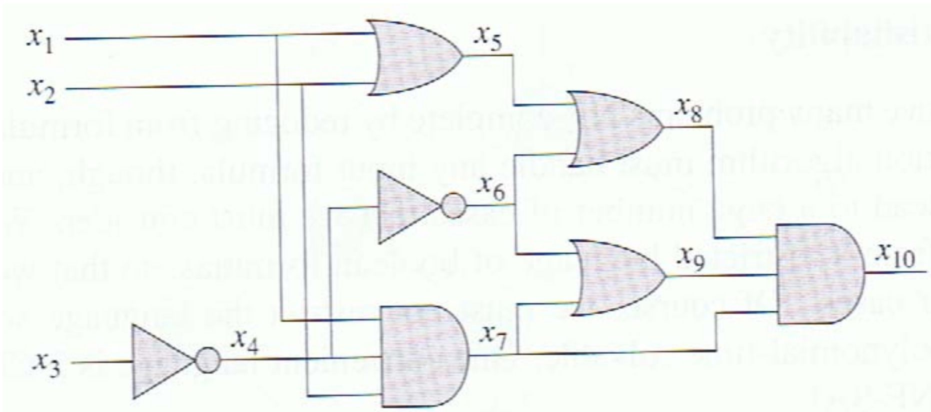
- SAT is NP-hard

- show that CIRCUIT-SAT $\leq_p$ SAT
- design a *reduction algorithm* that reduces any instance of circuit SAT to an instance of formula SAT in polynomial time

Reduction algorithm:

- assign a variable to each wire
- write a formula for each gate using the incident wires
- write the formula as a conjunction of clauses, describing the operation of each gate
- this alg is polynomial

Reduction algorithm, example



$$
\begin{aligned}
\phi = x_{10} &\wedge (x_4 \leftrightarrow \neg x_3) \\
&\wedge (x_5 \leftrightarrow (x_1 \vee x_2)) \\
&\wedge (x_6 \leftrightarrow \neg x_4) \\
&\wedge (x_7 \leftrightarrow (x_1 \wedge x_2 \wedge x_4)) \\
&\wedge (x_8 \leftrightarrow (x_5 \vee x_6)) \\
&\wedge (x_9 \leftrightarrow (x_6 \vee x_7)) \\
&\wedge (x_{10} \leftrightarrow (x_7 \wedge x_8 \wedge x_9)).
\end{aligned}
$$

SAT

- circuit C is satisfiable iff ϕ is satisfiable
- if C is sat, then each wire has a well defined value and the output of the circuit is 1
  - assign similar values to the variables in ϕ
  - each clause of ϕ evaluates to 1 $\Rightarrow$ conjunction of all evaluate to 1
- similarly, if ϕ has a satisfying assignment, then the corresponding circuit C evaluates to 1

3-CNF-SAT

- ***literal*** – variable or its negation
- ***conjunctive normal form (CNF)*** – a boolean formula expressed as an AND of clauses, and each clause is the OR of one or more literals
- ***3-CNF*** – if each clause has exactly three distinct literals

exp: $(x_1 \vee \neg x_1 \vee \neg x_2) \wedge (x_3 \vee x_2 \vee x_4) \wedge (\neg x_1 \vee \neg x_3 \vee \neg x_4)$

> ***3-CNF-SAT problem***: given a boolean formula ϕ in 3-CNF, is it satisfiable?

3-CNF-SAT

Theorem : 3-CNF-SAT is NP-complete.

Proof:

- 3-CNF-SAT $\in$ NP
- similar to SAT
- 3-CNF-SAT is NP-hard
- show that SAT $\leq_p$ 3-CNF-SAT
- reduction algorithm has 3 steps:

Step 1:

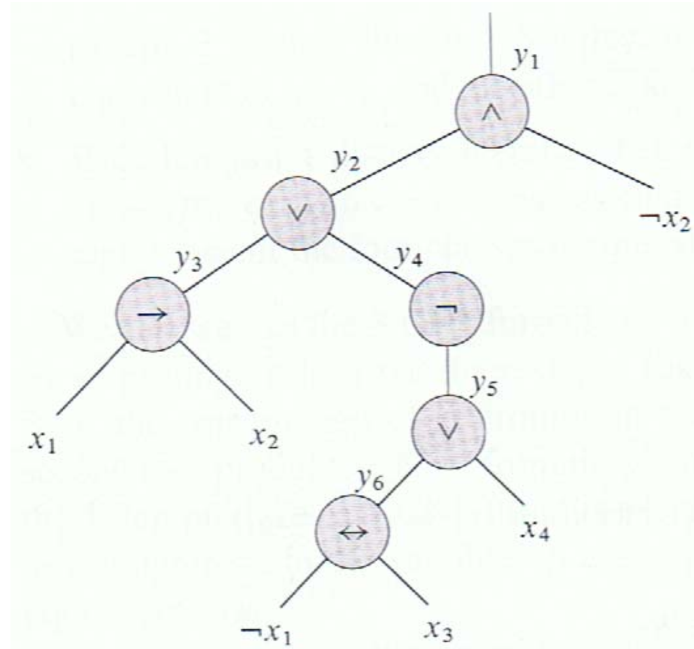
- construct a “binary” parse tree for the input formula ϕ, where literals are leaves and connectives are internal nodes
- assign variables y to output of each internal node
- rewrite ϕ as a conjunction of clauses describing the operation of each node

Example, step 1

Original formula:

$\phi = ((x_1 \rightarrow x_2) \vee \neg((\neg x_1 \leftrightarrow x_3) \vee x_4)) \wedge \neg x_2$

$$
\begin{aligned}
\phi' = y_1 &\wedge (y_1 \leftrightarrow (y_2 \wedge \neg x_2)) \\
&\wedge (y_2 \leftrightarrow (y_3 \vee y_4)) \\
&\wedge (y_3 \leftrightarrow (x_1 \rightarrow x_2)) \\
&\wedge (y_4 \leftrightarrow \neg\, y_5) \\
&\wedge (y_5 \leftrightarrow (y_6 \vee x_4)) \\
&\wedge (y_6 \leftrightarrow (\neg x_1 \leftrightarrow x_3))
\end{aligned}
$$



The requirement is not necessarily met:

- each clause must be the OR of 3 literals

3-CNF-SAT

Step 2:

- convert each clause ϕ'_i into CNF
- construct the truth table for ϕ'_i
- build the formula for $\neg\phi'_i$ in DNF (an OR of ANDs) by taking the table entries that evaluate to 0
- use DeMorgan's laws to convert to the CNF ϕ''

$$\neg(a \wedge b) = \neg a \vee \neg b$$

$$\neg(a \vee b) = \neg a \wedge \neg b$$

Example, step 2

$\phi'_1 = y_1 \leftrightarrow (y_2 \wedge \neg x_2)$

y_1	y_2	x_2	$(y_1 \leftrightarrow (y_2 \wedge \neg x_2))$
1	1	1	0
1	1	0	1
1	0	1	0
1	0	0	0
0	1	1	1
0	1	0	0
0	0	1	1
0	0	0	1

$\neg\phi'_1 = (y_1 \wedge y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge x_2) \vee (y_1 \wedge \neg y_2 \wedge \neg x_2) \vee (\neg y_1 \wedge y_2 \wedge \neg x_2)$

Applying the DeMorgan's laws:

$\phi''_1 = (\neg y_1 \vee \neg y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee \neg x_2) \wedge (\neg y_1 \vee y_2 \vee x_2)$
$\wedge (y_1 \vee \neg y_2 \vee x_2)$

Requirement not necessarily met:

- each clause must have exactly 3 literals

3-CNF-SAT

Step 3: for each clause C_i of ϕ'':

- if C_i has 3 distinct literals, then include C_i as a clause of ϕ'''
- if C_i has 2 distinct literals, e.g. $C_i = (l_1 \vee l_2)$ then include $(l_1 \vee l_2 \vee p) \wedge (l_1 \vee l_2 \vee \neg p)$ as clauses of ϕ'''

  regardless of the value of p, one clause becomes 1 and the other $(l_1 \vee l_2)$
- if C_i has 1 distinct literal, e.g. C_i = t, then include $(t \vee p \vee q) \wedge (t \vee p \vee \neg q) \wedge (t \vee \neg p \vee q) \wedge (t \vee \neg p \vee \neg q)$ as clauses of ϕ'''

  regardless of the values of p and q, one clause becomes t and all other 3 become 1.

3-CNF-SAT

- 3-CNF formula ϕ''' is satisfiable iff ϕ is satisfiable
- reduction algorithm is polynomial:
  - constructing ϕ' from ϕ introduces at most 1 variable and 1 clause per connective in ϕ
  - constructing ϕ'' from ϕ' introduces at most 8 clauses in ϕ'' from each clause in ϕ'
  - constructing ϕ''' from ϕ'' introduces at most 4 clauses into ϕ''' for each clause in ϕ''
- thus the size of the resulting formula ϕ''' is polynomial in terms of the length of the original formula
- each step construction is accomplished in polynomial time

Other NP-complete problems

- The clique problem
- The vertex cover problem
- The traveling salesman problem

Graph coloring problem

History

- map coloring: color a map with minimum number of colors, s.t. neighboring regions use different colors
  - e.g. color the states of the US map or the countries on the globe
- in the 19th century, Francis Guthrie noticed that you could color a map of the counties of England with only 4 colors, and wondered whether the same was true for any map
- he asked his brother who passed the question to his mathematics teacher, and thus a famous mathematical problem was born: the *Four-Color Conjecture*

Map coloring, cont.

- after more than 100 years, the conjecture was finally proved in 1976 by Appel and Haken
- the proof was an induction on the number of regions, but the induction step involved nearly 2000 cases, and the verification had to be carried on by a computer
- finding a reasonably short, human-readable proof still remains open

Graph coloring problem

- polynomial algorithm to find whether a graph G is 2-colorable
- 3-coloring is NP-complete
- for k > 3, k-coloring is NP-complete

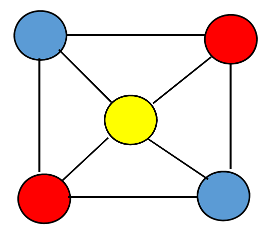
Reference on the graph coloring problem:

- Kleinberg & Tardos ch 8.7

Graph Coloring Problem

- Optimization problem: Given an undirected graph G(V, E), we seek to assign a color to each vertex such that for any edge $(u, v) \in E$, u and v are assigned different colors. The objective is to color G using a *minimum* number of colors.

- If G has a k-coloring, then we say that G is a k-colorable graph.

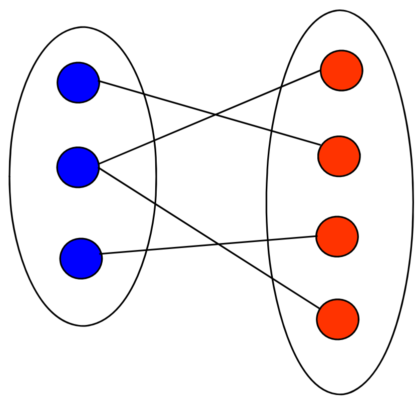


3-colorable graph

- Decision problem: Given an undirected graph G and a value k, does G have a k-coloring?

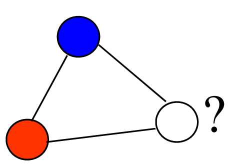
2-Coloring problem

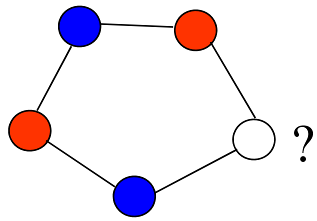
- A graph G is 2-colorable iff G is bipartite



-V is partitioned into two sets X and Y
- any edge (u,v) has one endpoint in X and the other in Y

- A graph G which is 2-colorable cannot contain an odd cycle





Breadth-First-Search (BFS)

CLRS page 595

- Given a graph G and a source vertex s $\in$ V, discover all vertices reachable from s.

For each vertex v, keep the following fields:

- v.d – distance (number of edges) on the shortest-path from s to v
- v.π – predecessor of v on the shortest-path from s to v
- v.color – one of the following:
  - ❖ WHITE – undiscovered
  - ❖ GRAY – discovered but not finished
  - ❖ BLACK – finished
  - - Note that this "color" is not related to the 2-coloring problem

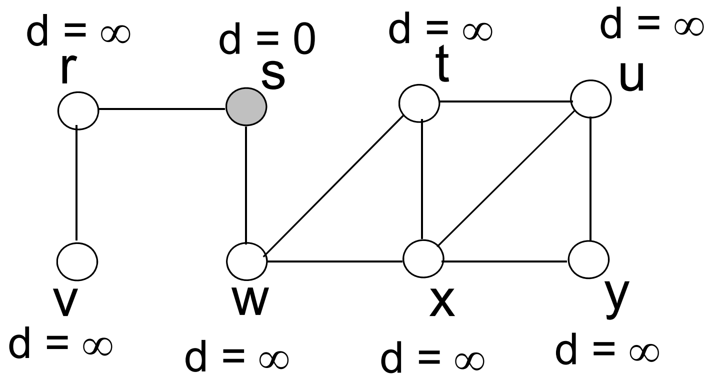
BFS

- Rule: discover all the vertices at distance k before discovering vertices at distance k+1

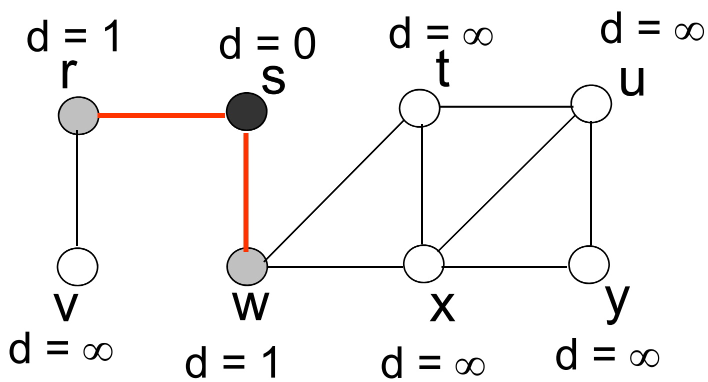
```
BFS(G, s)
for each vertex u ∈ G.V − {s}
    u.color = WHITE
    u.d = ∞
    u.π = NIL
s.color = GRAY
s.d = 0
s.π = NIL
Q = ∅
ENQUEUE(Q, s)
while Q ≠ ∅
    u = DEQUEUE(Q)
    for each v ∈ G.Adj[u]
        if v.color == WHITE
            v.color = GRAY
            v.d = u.d + 1
            v.π = u
            ENQUEUE(Q, v)
    u.color = BLACK
```

RT = $\Theta(|V| + |E|)$

BFS

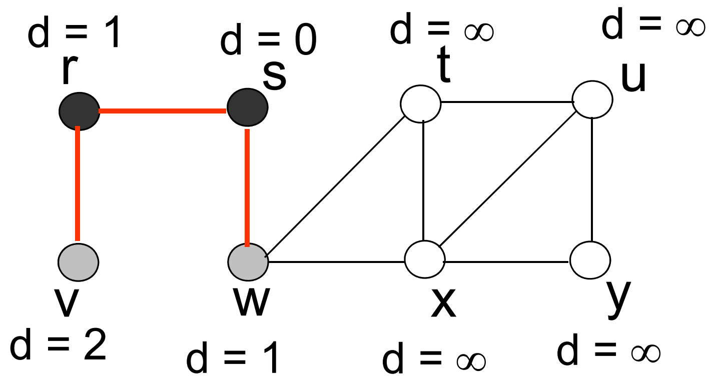


Q = $\boxed{s}$

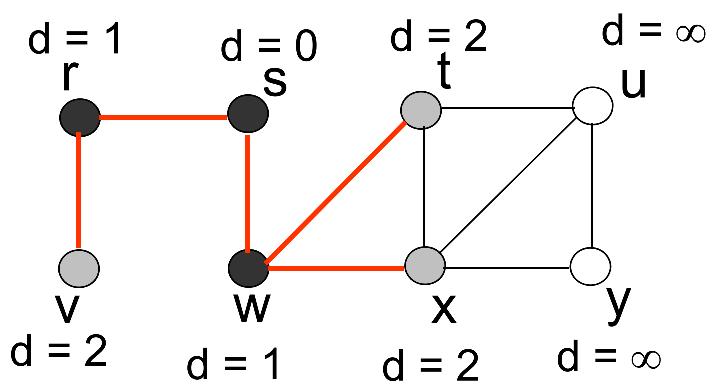


Q = $\boxed{\cancel{s}\ r\ w}$

BFS

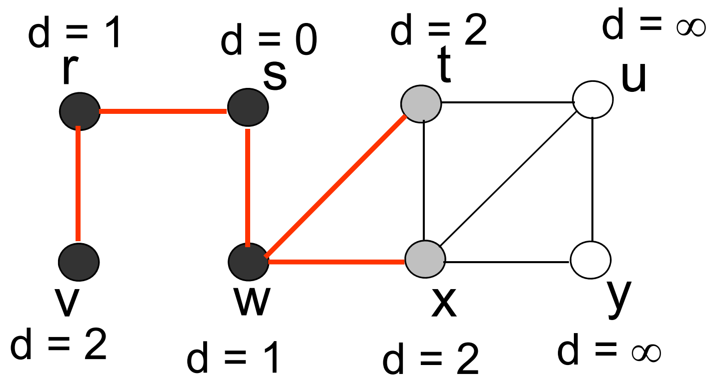


Q = ~~s~~ ~~r~~ w v

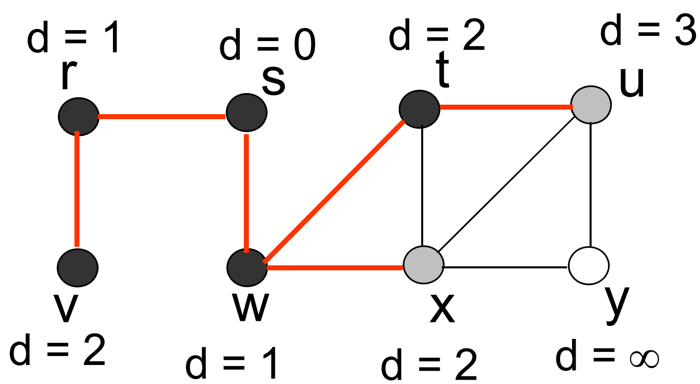


Q = ~~s~~ ~~r~~ ~~w~~ v t x

BFS

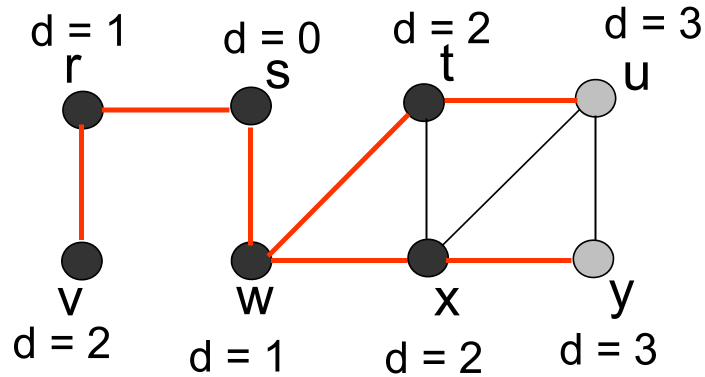


Q = ~~s~~ ~~r~~ ~~w~~ ~~v~~ t x

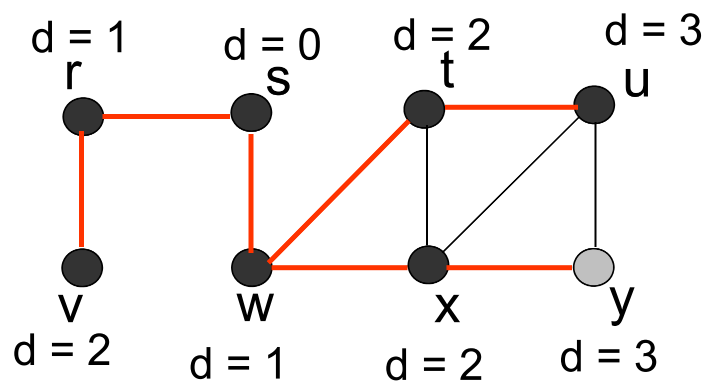


Q = ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ x u

BFS

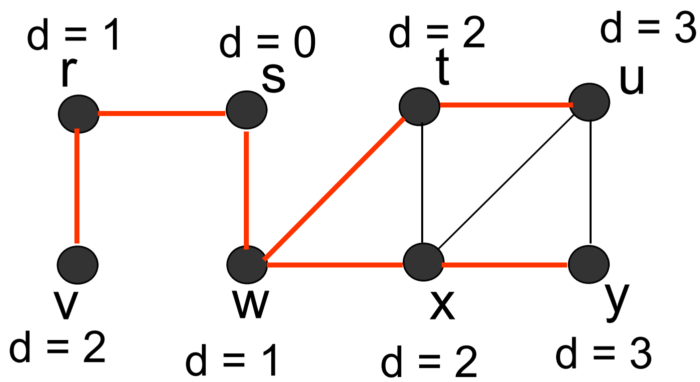


Q = $\boxed{\cancel{s}\ \cancel{r}\ \cancel{w}\ \cancel{v}\ \cancel{t}\ \cancel{x}\ u\ y}$



Q = $\boxed{\cancel{s}\ \cancel{r}\ \cancel{w}\ \cancel{v}\ \cancel{t}\ \cancel{x}\ \cancel{u}\ y}$

BFS



Q = ~~s~~ ~~r~~ ~~w~~ ~~v~~ ~~t~~ ~~x~~ ~~u~~ ~~y~~

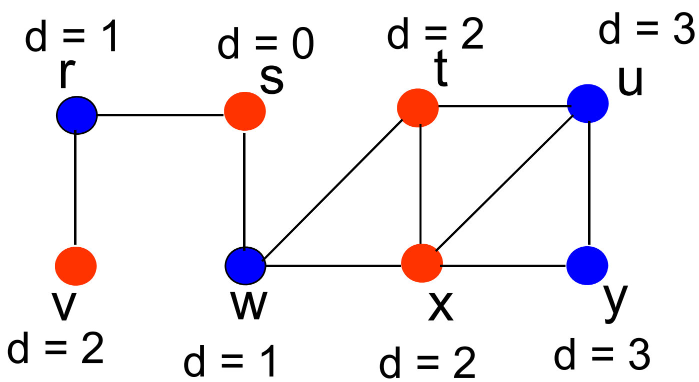
BFS – computes the shortest-paths from s to all vertices v reachable from s

2-coloring algorithm

- start from a vertex s
- run BFS
- color s red, all nodes at distance 1 blue, all nodes at distance 2 red, …so on
  - if the distance is an odd number: color = blue
  - if the distance is an even number: color = red
- scan all edges and check if both ends have received different colors
  - YES $\Rightarrow$ graph is 2-colorable
  - NO $\Rightarrow$ graph is NOT 2-colorable

RT = $\Theta(|V| + |E|)$

2-coloring algorithm, example



- The graph is NOT 2-colorable