Representation of polynomials

- Coefficient Representation
  - $A(x)= \sum a_j x^j$
- Point Value representation
  - $< y_0, y_1,\ldots,y_{n-1} >$ evaluated at $< x_0, x_1,\ldots,x_{n-1} >$
- Evaluation at given x
  - $A(x)= a_0+ x(a_1+ x(a_2+\ldots))\ldots) = \sum a_j x^j$
  - Choose $< x_0, x_1,\ldots,x_{n-1} >$ as the 2n-th roots of unity
  - $\omega^k{}_n = \exp(2\pi i\ k/n) = \cos(2\pi\ k/n) + i \sin(2\pi\ k/n)$

Operation on polynomials

Coefficient representation

- Addition - $O(n)$
  - $C(x)=A(x)+B(x)$
    - $C[j]=a[j]+b[j]$
- Multiplication - $O(n^2)$
  - $C(x)=A(x) \circ B(x)$
    - $C[j] = \sum a[k]b[j-k]$
    - convolution
- Transform to point value
    - $> y = V . a$

Point value representation

- Addition - $O(n)$
  - $C(x)=A(x)+B(x)$
    - $< y_{c,i} > \; = \; < y_{a,i} + y_{b,i} >$
- Multiplication - $O(n)$
  - $C(x)=A(x) \;\; B(x)$
    - $< y_{c,i} > \; = \; < y_{a,i} \, . \, y_{b,i} >$
    - element wise
- Transform to coefficient
    - $> a = V^{-1} . y$

Properties of roots of unity

- Group under multiplication: $\omega^k_n \, \omega^j_n = \omega^{k+j}_n$
- Cancellation: $\omega^{dk}_{dn} = \omega^k_n$
- Squaring: $(\omega^{k+n/2}_n)^2 = \omega^{2k}_n \omega^n_n = (\omega^k_n)^2 = (\omega^k_{n/2})$
  - Squares of n complex n-th roots = n/2 complex n/2-th roots
- Summing all roots: $\sum(\omega^k_n)^j = ((\omega^k_n)^n - 1)/(\omega^k_n - 1)) = 0$
- (k,j) th entry of V is (ω^{kj}_n)
- (j,k) th entry of V^{-1} has to be $(\omega^{-kj}_n)/n$, shown below
- $[V^{-1} V]_{jj'}$ is $\sum (\omega^{-kj}_n/n)(\omega^{kj'}_n) = \sum (\omega^{k(j'-j)}_n/n)$
- When j=j', $[V^{-1} V]_{jj'} = 1$; 0 otherwise so that $[V^{-1} V] = I$

Discrete Fourier Transform

- $< y_0, y_1,\ldots,y_{n-1} > = \mathrm{DFT}\ (a_0, a_1,\ldots,a_{n-1})$
- $y_k = \sum a_J\ (\omega^{kj}{}_n)$ with $A(x)= \sum a_j\, x^j$ and $x= \omega^{kj}{}_n$
- $A^{[0]}(x)= a_0 + a_2\, x+ a_4\, x^2+ \ldots + a_{n-2}\, x^{n/2-1}$
- $A^{[1]}(x)= a_1 + a_3\, x+ a_5\, x^2+ \ldots + a_{n-1}\, x^{n/2-1}$
- $A(x) = A^{[0]}(x^2) + x\, A^{[1]}(x^2) \rightarrow$ divide and conquer
- Evaluating $A^{[0]}(x^2)$ at $\omega^k{}_n$ $\Rightarrow$ Evaluating $A^{[0]}(x)$ at $\omega^k{}_{n/2}$
- Therefore problem splits into two equal subproblems
- $T(n)=2\ T(n/2) + O(n) \rightarrow T(n) = O(n \lg n)$

Recursive FFT algorithm (a)

- Basis: if n==1 return a // n=length[a] = power of 2
- Initialize: $\omega_n = \exp(2\pi i/n)$ and $\omega=1$
- Recursive DFT:
  - $y^{[0]} = \mathrm{RFFT}(a^{[0]}) \Rightarrow y^{[0]}_k = A^{[0]}(\omega^k_{n/2}) = A^{[0]}(\omega^{2k}_n)$
  - $y^{[1]} = \mathrm{RFFT}(a^{[1]}) \Rightarrow y^{[1]}_k = A^{[1]}(\omega^k_{n/2}) = A^{[1]}(\omega^{2k}_n)$
- Combine results
  - For k= 0 to n/2 -1
  - $y_k = y^{[0]}_k + \omega y^{[1]}_k$; $y_{k+n/2} = y^{[0]}_k - \omega y^{[1]}_k$
- Update $\omega = \omega\, \omega_n$
- Return column vector y
- Inverse DFT is same problem with y replacing a

Number Theoretic Algorithms

- Problem size is linear => proportional to number of bits needed to store the number in binary
- $O(n)$ for number n is exponential complexity
- **Fast exponentiation** – x^n – linear in width of n
  - Convert n to binary
  - Compute successive squares of x (takes $O(\lg n)$)
  - Use binary string to pick relevant powers of x
  - Example: $3^{11} \bmod 20 = (1*9*3) \bmod 20 = 7$
  - $(3^8 \bmod 20)\ (3^2 \bmod 20)(3^8 \bmod 20)$

GCD computation

euclid (a,b)

If b==0 then return a

Else return (euclid(b, a mod b))

Correctness: r_{m+2} is gcd (a,b)

$a=q_1b+r_1$, $b=q_2r_1+r_2$, $r_1=q_3r_2+r_3$; $r_i=q_{i+2}r_{i+1}+r_{i+2}$

$r_{m-1}=q_{m+1}r_m+r_{m+1}$; $r_m=q_{m+2}r_{m+1}+r_{m+2}$; $r_{m+1}=q_{m+3}r_{m+2}+0$

Now r_{m+2} divides successively r_{m+1} , r_m, r_{m-1}, a, b

Complexity: gcd(a,b) -> gcd(b,c) -> gcd(c,d) implies b = kc + d with at least k=1; means b ≥ c+d and together with a > b gives a+b > 2(c+d). Therefore in every two steps, the sum of the numbers get halved. Hence number of steps (or calls) would be bounded by log (b) the smaller one.

Worst case: The Fibonacci sequence i.e. $a=F_{n+1}$ and $b= F_n$

Extended Euclid algorithm

Express GCD (a,b) as linear combination of a,b
Say, d=gcd(a,b) = ax +by -> to find integers x, y

Example: a=289, b=204 => gcd=17 so that x=5 and y= -7

Extended_euclid (a,b)

```
If b==0
    return (a,1,0)
(d',x',y') = Extended_euclid(b, a mod b)
(d'',x'',y'') = (d', y', x' – floor(a/b) y')
```

Apply this to (a.b) mod n =1; a,b are multiplicative inverses mod n.
Example: Find multiplicative inverse of 50 mod 71 is 27,
50 and 71 are relatively prime, since gcd(50,71)=1

Primality testing

- Complexity is normally exponential actually O(√n)
- Can be reduced to linear time – approximation

Fermat's theorem: If p is prime, a is +ve integer,
a^{p-1} mod p = 1 i.e. a and p are relatively prime

```
Is_prime(p)
    choose a random no a such that 1 < a ≤ p-1
    compute x =(a^(p-1) ) mod p [fast exponentiation]
    if x ≠ 1 , p is composite
    else repeat several times, using different a's
```

Due to existence of pseudo-primes, condition may fail.

Pseudo-primes

- Number composite, yet obeys $a^{p-1} \bmod p = 1$ for certain choice of a – Base-a pseudoprime
- Base-2 pseudoprime: 341, 561, 645, 1105
- Carmichel number: 561, 1105, 1729 (all bases)
- Distribution of prime numbers:
  - $\pi(n)$ = no of primes $\leq n$
  - $\mathrm{Lt}_{n \to \mathrm{INF}} [\pi(n) / (n/\ln n)] = 1$
  - Implies that density of primes is $(n/\ln n)$

Miller Rabin test for primality

WITNESS (a,n)
  $x_0 \leftarrow a^d$ mod n
  for i=1 to *r*
    $x_i \leftarrow x_{i-1}{}^2$ mod n
    if x_i = 1 and $x_{i-1} \neq 1$ *and* $x_{i-1} \neq n-1$
      return *TRUE*
  if $x_t \neq 1$ **then return** *TRUE*
  return *FALSE*

MillerRabin (n,s)
for j=1 to s
  a=RANDOM(1,n-1)
  if WITNESS(a,n) return Composite
return *probably prime*

- If there exists a nontrivial square root of 1, modulo n, then n is composite e.g. take 6^2 (mod 35) =1 but √1 ≠ 6.
- When p^e divides (x^2 -1) i.e. (x-1) or (x+1) so that x^2 is 1 (mod p^e) which yields solution trivially 1.
- While computing each modular exponentiation, Miller Rabin test looks for a nontrivial square root of 1, modulo n during the squaring or power raisings.
- If it finds one, it stops and returns COMPOSITE. This way it fools 561, Carmichel number (shown for a=7)

Example: How Miller Rabin test works

- Take p=1729; n=p-1= 27.2.2.2.2.2.2 i.e. r=6, d=27
- Pick a=11 (randomly)
- Applying modular exponentiation a^d (mod n) gives sequence (11, 121, 809, 919, 809)
- This is followed by the sequence upon squaring of 11^27 giving (1331, 1065, **1**, ...)
- Existence of the **1** in this latter sequence shows the presence of non-trivial square root of 1.
- Otherwise 11^1728 mod 1729 = 1 implies prime.

Public key cryptosystem

Public key used is P; Private key used is S

- **Message:**
  - Message encrypted – P of receiver by sender
  - Message decrypted – S of receiver used
  - Cyphertext $C = P(M)$ used for encryption
  - $M = S(C) = S(P(M))$ to decrypt the message
- **Signature:**
  - Signature encrypted with own S by sender
  - Signature decrypted by recipient with P of sender
  - Signature encrypted using $\Sigma = S(\sigma)$
  - Signature decrypted using $\sigma = P(\Sigma) = P(S(\Sigma))$

Creation of public and private keys

- Select at random two large primes p and q
- Compute n = pq
- Select small odd integer e relatively prime to $\Phi(n) = (p-1)(q-1)$
- Compute d as multiplicative inverse of e modulo $\Phi(n)$ i.e. d.e mod (p-1)(q-1) = 1.
- Publish P= (e,n) as public key
- Publish S= (d,n) as private key

RSA cryptosystem Protocol

- $P(M) = C = M^e \pmod{n}$
  - cyphertext created using public key of recipient, decryption would need private key of the intended recipient
- $S(C) = C^d \pmod{n}$
  - signed using private key of sender, verify with sender public key
- If p,q are 256 byte numbers, n is 512 bytes.
- $P(S(M)) = S(P(M))= M^{ed} \pmod{n}$; $n=pq$
- Now $ed = 1 + k(p-1)(q-1)$
- Hence $M^{ed} \pmod{n} = M(M^{p-1})^{k(q-1)} \pmod{n}$
- Using Fermat's theorem, $M^{p-1} \pmod{p} = 1$
- Hence $M^{ed} \pmod{n} = M \pmod{p} = M \pmod{q}$
- Chinese Remainder Theorem $M^{ed} = M \pmod{pq}$

RSA cryptosystem Protocol

- AA (d1,e1,n1) sends M to BB (d2,e2,n2)
- AA Encrypts Y1= M^{e2}(mod n2)
- AA Signs Y2= $Y1^{d1}$(mod n1)
- AA transmits Y2
- BB verifies sign Z1= $Y2^{e1}$(mod n1)
- BB decrypts Z2= $Z1^{d2}$(mod n2)
- Encrypt-Sign-Transmit-Verify sign-Decrypt

Attacking RSA cryptosystem

- Find a number that leaves remainder 2 when divided by 3 (p) and 3 when divided by 5(q)
  - – Chinese Remainder Theorem
- Then n= 2 (mod 3) and n= 3 (mod 5) gives n=8
- Such number is unique in the domain [1 .. pq]
- n and e are known, can we find d?
- We need to know Φ(n) from n
- Then we need to factorize n into its prime factors – but integer factorization is hard.

Single Source Shortest Path Problem

- Given a directed graph G = (V,E), with non-negative costs on each edge, and a selected source node v in V, for all w in V, find the cost of the least cost path from v to w.
- The *cost* of a path is simply the sum of the costs on the edges traversed by the path.
- This problem is a general case of the more common subproblem, in which we seek the least cost path from v to a particular w in V. In the general case, this subproblem is no easier to solve than the SSSP problem.

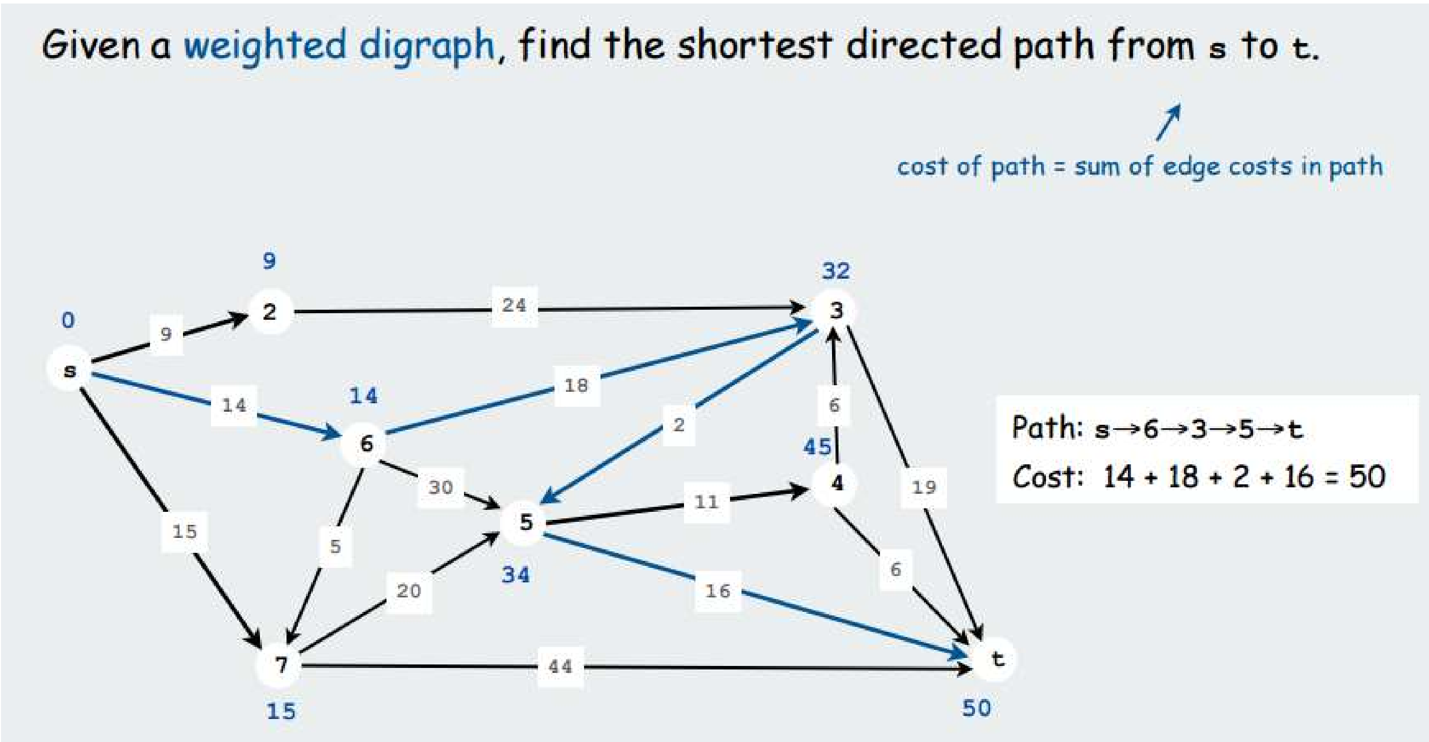
Dijkstra's Algorithm

- Dijkstra's algorithm is a *greedy* algorithm for the SSSP problem.
- A "greedy" algorithm always makes the locally optimal choice under the assumption that this will lead to an optimal solution overall.
- Data structures used by Dijkstra's algorithm include:
- a cost matrix C, where C[i,j] is the weight on the edge connecting node i to node j. If there is no such edge, C[i,j] = infinity.
- a set of nodes S, containing all the nodes whose shortest path from the source node is known. Initially, S contains only the source node.
- a distance vector D, where D[i] contains the cost of the shortest path (so far) from the source node to node i, using only those nodes in S as intermediaries.

How Dijkstra's Algorithm Works

- On each iteration of the main loop, we add vertex w to S, where w has the least cost path from the source v (D[w]) involving only nodes in S.
- We know that D[w] is the cost of the least cost path from the source v to w (even though it only uses nodes in S).
- If there is a lower cost path from the source v to w going through node x (where x is not in S) then
  - D[x] would be less than D[w]
  - x would be selected before w
  - x would be in S

SSSP - One illustration



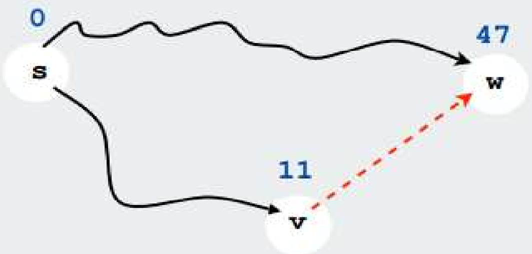
Edge relaxation

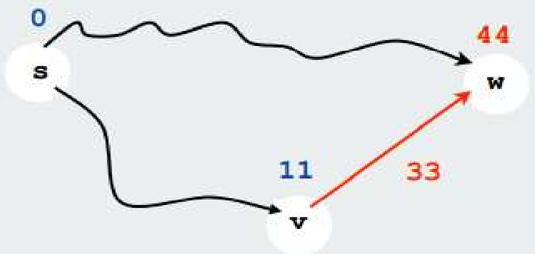
For all v, dist[v] is the length of some path from s to v.

Relaxation along edge e from v to w.

- dist[v] is length of some path from s to v
- dist[w] is length of some path from s to w
- if v-w gives a shorter path to w through v, update dist[w] and pred[w]

```
if (dist[w] > dist[v] + e.weight())
{
    dist[w] = dist[v] + e.weight());
    pred[w] = e;
}
```





Relaxation sets dist[w] to the length of a shorter path from s to w (if v-w gives one)

Djikstra’s Algorithm

S: set of vertices for which the shortest path length from s is known.

Invariant: for v in S, dist[v] is the length of the shortest path from s to v.

Initialize S to s, dist[s] to 0, dist[v] to ∞ for all other v

Repeat until S contains all vertices connected to s

- find e with v in S and w in S' that minimizes dist[v] + e.weight()
- relax along that edge
- add w to S

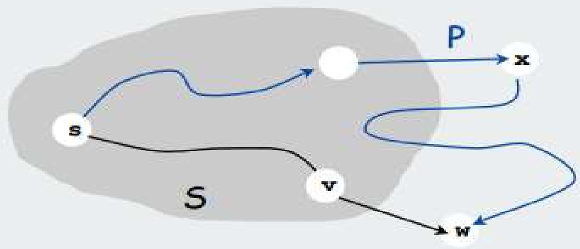
Correctness of the algorithm

S: set of vertices for which the shortest path length from s is known.

Invariant: for v in S, `dist[v]` is the length of the shortest path from s to v.

Pf. (by induction on |S|)

- Let w be next vertex added to S.
- Let P* be the s–w path through v.
- Consider any other s–w path P, and let x be first node on path outside S.
- P is already longer than P* as soon as it reaches x by greedy choice.



Analysis of Dijkstra's Algorithm

- Consider the time spent in the two loops:
- The first loop has O(N) iterations, where N is the number of nodes in G.
- The second (and outermost) loop is executed O(N) times.
  - The first nested loop is O(N) since we examine each vertex to determine whether or not it is in V-S.
  - The second nested loop is O(N) since we examine each vertex to determine whether or not it is in V-S.
- The algorithm is O(N^2).
- If we assume that there are many fewer edges than the maximum possible, we can do better than this.

Complexity of algorithms

- Polynomial time: worst case O(n^k)
- Super polynomial time: solvable but not in Polynomial time
- Unknown status: no Polynomial time algorithm found, no proof of Super Polynomial time lower bound

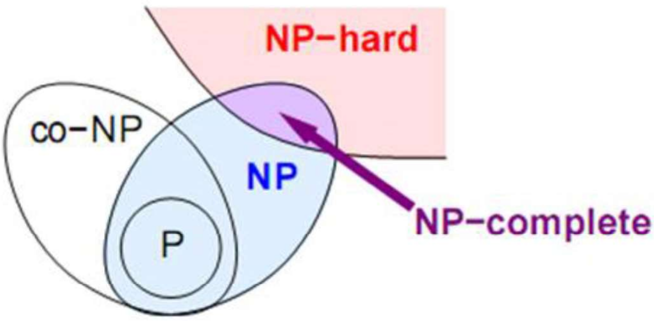
Models of computation

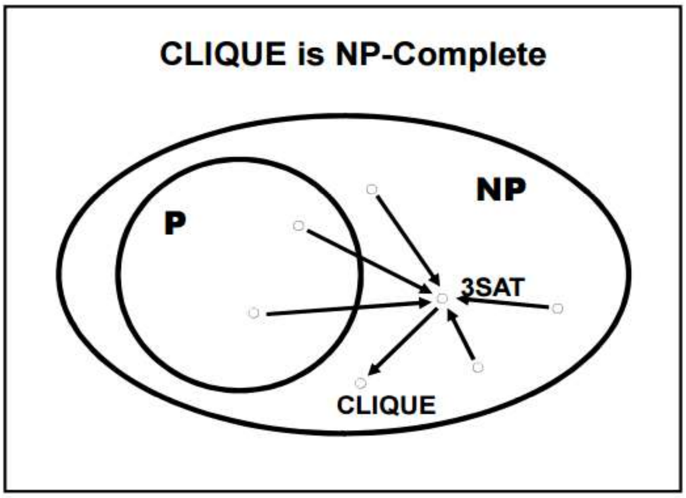
- Serial random access machine
- Parallel random access machine
- Abstract Turing machine

Complexity Classes

- P: problems that can be solved in *polynomial time* (typically in *n*, size of input) on a *deterministic Turing machine (DTM)*
  - Any normal computer simulates a DTM
- NP: problems that can be solved in *polynomial time* on a *non-deterministic Turing machine (NDTM)*
  - Informally, if we could "guess" the solution, we can *verify* the solution in P time (on a DTM)
  - NP does NOT stand for *non-polynomial*, since there are problems harder than NP
  - P is actually a subset of NP (we think)

Complexity Classes Overview





- NP-hard
  - At least as hard as any known NP problem (could be harder!)
  - Set of interrelated problems that can be solved by *reducing* to another known problem
- NP-Complete
  - A problem that is in NP and NP-hard
- Cook's Theorem
  - SATISFIABILITY (SAT) is NPC
- Other NPC problems
  - Reduce to SAT or previous reduced problem

Problem definitions

- Abstract problem Q is a binary relation on a set I of problem instances and set S of solutions
- G=(V,E) : Instances of shortest path
- Solution: sequence of vertices for abstract problem
- Decision problem: I --> {0,1} is a given path of length less than some threshold
- The decision problem is as complex as the abstract problem
- The abstract problem is optimization problem and decision problem somehow maps it to binary.

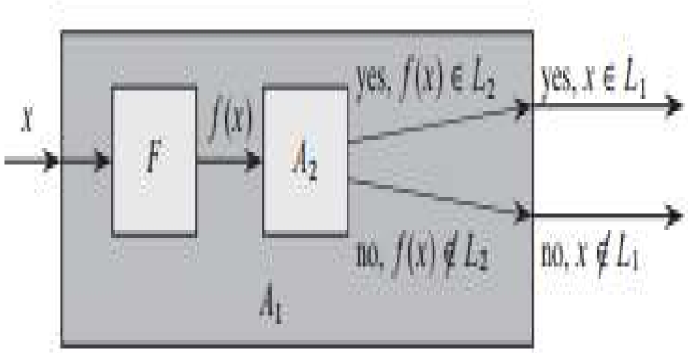
Problem encoding

- Unary encoding – integer k represented using k ones – somewhat like Roman number system
- Binary encoding – length n = floor(lg k)
- Linear complexity in unary encoding => log complexity in binary encoding
- Linear complexity in binary encoding => exponential complexity in unary encoding
- Two encodings e_1 and e_2 are related polynomially
- $f_{12}(e_1(i))=e_2(i)$ and $f_{21}(e_2(i))=e_1(i)$

P-time Reducibility

- Let Q be an abstract decision problem on an instance set I and encodings e_1 and e_2 are related polynomially on I – then e_1 (Q) ε P iff e_2 (Q) ε P
- If a problem Q reduces to another problem Q' then Q is no harder to solve than Q'
- Language L_1 is P-time reducible to L_2 denoted as $L_1 \leq_P L_2$ implies there is a P-time computable reduction function f:{0,1}* → {0,1}* such that for all x ε {0,1}* we have x ε L1 iff f(x) ε L2
- Hence for $L_1 \leq_P L_2$ then L_2 ε P implies L_1 ε P

Reduction mapping



- The algorithm F is a reduction algorithm
- Computes the reduction function f from L_1 to L_2 in P-time
- A_2 is P-time algorithm that decides L_2.
- A_1 decides whether x ε L_1 by using F to transform any input x into f(x) and then using A_2 to decide whether f(x) ε L_2

Notion of NP Completeness

- Class P – if there exists an algorithm A that decides L in polynomial time then L ε P
- For 2-input P-time algorithm A and constant c, L= {x ε {0,1}*: there exists a certificate y with |y| = $O(|x|^c)$ such that A(x,y)=1}
- If algorithm A verifies language L in P-time, then L ε NP
- In case L ε P then L ε NP (solved => verifiable)
- Property-1: L ε NP; Property-2: L' $\leq_P$ L for every L' ε NP
- When both properties hold then L ε NPC
- If only property-2 holds, then L ε NP-Hard

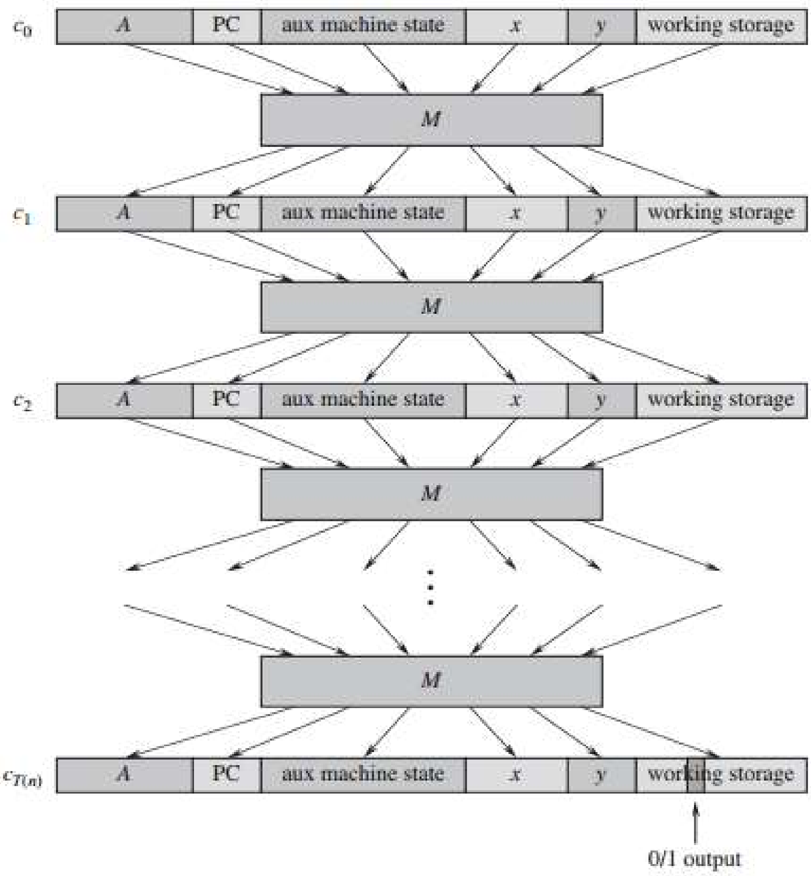
Theorem of NP-Completeness

- If any NPC problem is P-time solvable, P=NP
  - Suppose L belongs to both P and NPC. For any L' in NP, L' $\leq_P$ L (property-2 of NPC) Then such L' also belongs to P (reduce and solve)
- If any problem in NP is not P-time solvable, then all NPC problems are not P-time solvable
  - Suppose some L belongs to NP but not in P. Let L' be some NPC and to contradict assume that L' is in P. Then L $\leq_P$ L' (reduce) and hence L is also in P.

Circuit Satisfiability Problem

- Take any algorithm that produces output for given input in P-time => verification => εNP
- Can map this into program steps
- Each program step maps to combinational circuit
- Paste these circuits - maps to overall circuit
- Program I/O maps to circuit I/O in P-time
- All such algorithms are reducible ($\leq_P$) to CSAT
- CSAT is therefore NPC
- Such proof outline possible only for CSAT

Algorithm as computation sequence



Proof of NP-completeness for some L

- Prove L belongs to NP (verification decision)
- Select a known NP complete language L'
- Describe an algorithm that computes f, which is a function mapping every instance of L' to L
- Prove – for all x, f satisfies $x \varepsilon L'$ iff $f(x) \varepsilon L$
- Prove – algorithm computing f runs in P-time
- CSAT $\leq_P$ FSAT $\leq_P$ 3CNF-SAT $\leq_P$ CLIQUE (graph) $\leq_P$ VERTEX-COVER $\leq_P$ SUBSET-SUM (0-1 knapsack)
- CSAT $\leq_P$ FSAT $\leq_P$ 3CNF-SAT $\leq_P$ HC $\leq_P$ TSP

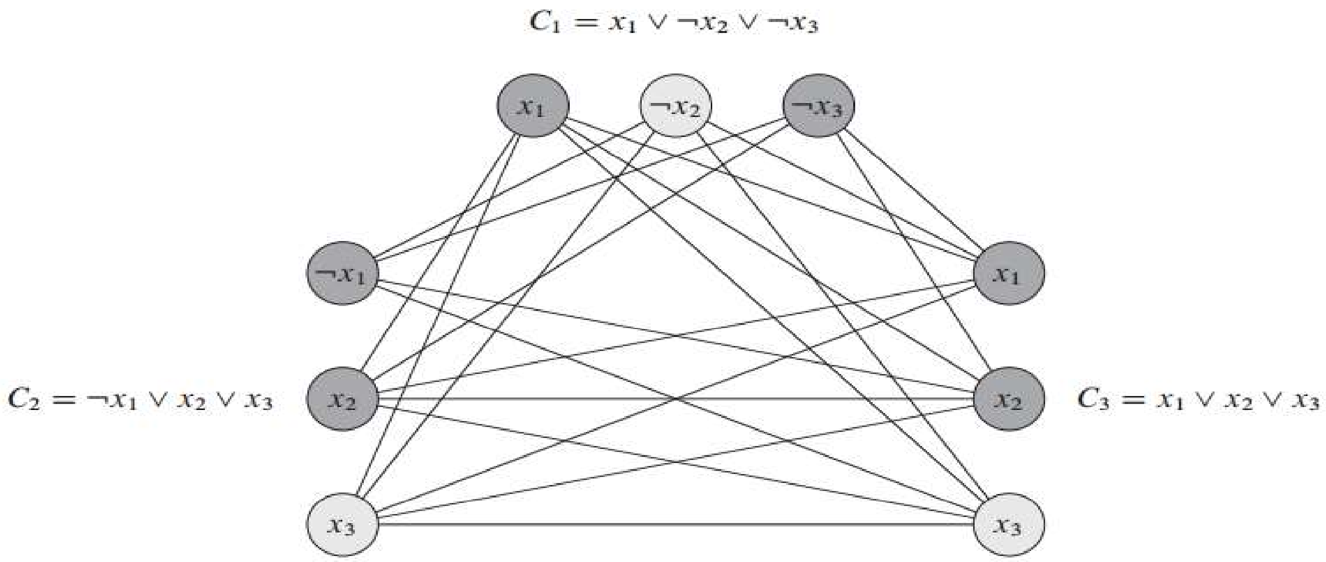
Clique of a graph

- Clique of size k: Find a subset of k vertices that are connected through edges to be found in E.
- Brute force - to check clique in all |k| subsets – runs in superpolynomial time C(|V|,k)
- Start from 3CNF: Boolean formula with k AND-ed clauses, each clause having 3 OR-ed literals
- The graph should be such that the formula is satisfiable iff the graph has a clique of size k

Reduction:- 3CNF-SAT $\leq_P$ CLIQUE

- For each clause C_r place triple vertices v_1r, v_2r, v_3r for the 3 literals l_1r, l_2r, l_3r
- Construct an edge between v_ir and v_ js when they fall in different clauses (r and s) AND their corresponding literals l_ir and l_ js are consistent i.e. l_ir not negation of l_ js
- When formula has a satisfying assignment, each clause has at least one 1 (OR within each clause and k such AND-ed clauses) resulting in k vertices. They form a clique since edges can be found by way of above construction.
- Conversely suppose G has a clique of size k. Since no edges in V connect same triple, this set has one vertex (read literal) per triple (read clause). We can assign 1 to each such literal since G has no edge between inconsistent literals. So each clause gets satisfied and the formula also gets satisfied.

Example: 3-CNF formula to graph



Clique mapping: The graph G derived from the 3-CNF formula $\phi = C_1 \wedge C_2 \wedge C_3$, where $C_1 = (x_1 \vee \neg x_2 \vee \neg x_3)$, $C_2 = (\neg x_1 \vee x_2 \vee x_3)$, and $C_3 = (x_1 \vee x_2 \vee x_3)$, in reducing 3-CNF-SAT to CLIQUE. A satisfying assignment of the formula has $x_2 = 0$, $x_3 = 1$, and x_1 either 0 or 1. This assignment satisfies C_1 with $\neg x_2$, and it satisfies C_2 and C_3 with x_3, corresponding to the clique with lightly shaded vertices.

Hallmarks of the CLIQUE mapping

- The graph constructed here is of a special kind since vertices here occur as triplets with no edges between vertices in same triplet
- This CLIQUE happens in restricted case but the corresponding 3CNF case is very general
- But if we had a polynomial-time algorithm that solved CLIQUE on general graphs, it would also solve CLIQUE on restricted graphs.
- Opposite approach is however not enough – in case an easy 3CNF instance were mapped, the NP-hard problem does not get mapped
- The reduction uses instances, not the solution – actually we do not know whether we can decide 3CNF-SAT in P-time!

Intuitive Mapping: 3CNF-SAT to HC

- One approach is to follow the reduction from clique: CLIQUE $\leq_P$ VERTEX-COVER $\leq_P$ $\leq_P$ Hamiltonian Path $\leq_P$ HC
- Direct approach - Encode an instance I of 3-SAT as a graph G such that I is satisfiable exactly when G has HC
- Create some graph that represents the variables
- Create some graph that represents the clauses (each clause has exactly three literals/variables)
- Hook up the variables with the clauses such that the formula gets encoded
- Show that this graph has HC iff the formula in conjunctive normal form is satisfiable.

Reduction – HC to TSP

- Travelling Salesman Problem works on complete graph G=(V,E) with cost function c defined from VxV -> Z with k ε Z and G has a TSP tour with cost at most k
- Let G=(V,E) be an instance of HC. Form the complete graph G'=(V,E') with cost function $c(V_i,V_j)$ with k=0
- c(i,j) = 0 if (i,j) ε E and c(i,j) = 1 if (i,j) ε E'-E
- Instance of TSP is taken to be TSP(G',c,0)
- Since G has HC, G' has valid TSP tour of at most 0
- If G' has TSP tour of 0, the tour contains only edges from E, which in turn implies that G has HC => HC $\leq_P$ TSP (NP-hard)
- Since a TSP tour can be verified in P-time it is in NP and together with it being NP-hard therefore TSP ε NPC.

Branch & Bound approximation for TSP

- B&B algorithm performs a top-down recursive search through the tree of instances formed by the branch operation.
- Upon visiting an instance *I*, it checks whether bound(*I*) is greater than the upper bound for some other instance that it already visited; if so, *I* may be safely discarded from the search and the recursion stops.
- This pruning step is usually implemented by maintaining a global variable that records the minimum upper bound seen among all instances examined so far.
- Generic B&B is related with backtracking as in DFS traversals
- And the journey continues...