Author

Xubo Luo

1. Stable Matching

Gale-Shapley algorithm

```
Gale-Shapley(preference lists ofr hospitals and students):
    Initialize M to empty matching.
    while(some hospital h is unmatched and has not proposed to every student):
        s <- first student on h's list to whom h has not yet proposed
        if(s is unmatched):
            Add h-s to matching M
        elif(s prefer h to current partner h*):
            Replace h*-s with h-s in matching M
        else:
            s rejects h
    return stable matching M.
```

2. Algorithm Analysis

Big O notation

Upper bounds. f(n) is O(g(n)) if there exist constants c > 0 and n_0 >= 0 such that 0 <= f(n) <= c*g(n) for all n >= n_0

Big Omega notation

Lower bounds. f(n) is Ω(g(n)) if there exist constants c > 0 and n_0 >= 0 such that f(n) >= c*g(n) >= 0 for all n >= n_0

Big Theta notation

Tight bounds. f(n) is Θ(g(n)) if there exist constants c1>0, c2>0, and n_0 >= 0 such that 0<=c1 * g(n) <= f(n) <= c2 * g(n) for all n >= n_0

3.Graphs

Basic definitions and applications

Trees

Def. An undirected graph is a *tree* if it is connected and does not contain a cycle.

Theorem. Let G be an undirected graph on n nodes. Any two of the following statements imply the third:

- G is connected
- G does not contain a cycle
- G has n-1 edges

Graph connectivity and graph traversal

BFS, DFS - O(m+n)

- BFS = explore in order of distance from s
- DFS = explore in a different way

Testing bipartiteness

Lemma. If a graph G is bipartite, it cannot contain an odd-length cycle

Lemma. If G be a connected graph, and let L0, ..., Lk be the layers produced by BFS starting at node s. Exactly one of the following holds.

1. No edges of G joins two nodes of the same layer, and G is bipartite
2. An edge of G joins two nodes of the same layer, and G contains an odd-length cycle(and hence is not bipartite)

Corollary. A graph G is bipartite `iff` it contains no odd-length cycle.

Connectivity in directed graphs

Def. A graph is *strong connected* if every pair of nodes is mutually reachable.

- Use BFS on G and $G^{reverse}$ to determine if G is strongly connected.

Def. A *strong component* is a maximal subset of mutually reachable nodes.

DAGs and topological ordering

Lemma. G is a DAG `iff` G has a topological ordering.

4.Greedy Algorithm

Coin changing

```
cashier algorithm(x,c1,c2,..., cn):
    sort n coin denominations so that 0 < c1 < c2 <... < cn
    S = []
    while(x>0):
        k = largest coin denomination ck such that ck <= x
        if(no such k):
            return 'no solution'
        else:
            x = x-ck
            S = S + {k}
    return S
```

Interval scheduling

```
earliest finish time first(n, s1, s2, ..., sn, f1, f2, ..., fn):
    sort jobs by finish times and renumber so that f1<=f2<=...<=fn
    S = []
    for j = 1 to n:
        if(job j is compatible with S):
            S = S + {j}
    return S
```

Interval partitioning

- Lecture j starts at s_j and finishes at f_j
- Goal: find minimum number of classrooms to schedule all lectures so that no two lectures occur at the same time in the same room.

```
earliest start time first(n, s1, s2,..., sn, f1, f2, ..., fn):
    sort lectures by start times and renumber so that s1<=s2<=...<=sn
    d = 0
    for j =1 to n:
        if (lecture j is compatible with some classroom):
            schedule lecture j in any such classroom k
        else:
            allocate a new classroom d+1
            schedule lecture j in classroom d+1
            d = d+1
    return schedule
```

Scheduling to minimize lateness

- Single resource processes one job at a time
- Job j requires tj units of processing time and is due at time dj
- If j starts at time sj, it finishes at time fj = sj + tj
- Lateness: αj = max(0, fj-dj)
- Goal: schedule all jobs to minimize maximum lateness L = $\max_j$ αj

```
Earliest deadilne first(n, t1, t2, ..., tn, d1, d2, ..., dn):
    sort jobs by due times and renumber so that d1 <= d2 <= ... <= dn
    t = 0
    for j = 1 to n:
        assign job j to interval[t,t+tj]
        sj = t
        fj = t + tj
        t = t + tj
    return intervals[s1,f1,],[s2,f2], ..., [sn,fn]
```

Optimal caching

Caching

- Cache with capacity to store k items
- Sequence of m item requests d1,d2,...,dm
- Cache hit: item in cache when requested
- Cache miss: item not in cache when requested

Goal. Eviction schedule that minimize the number of evictions.

Dijkstra's algorithm

For single-source shortest paths problem

```
dijkstra(V,E,len,s):
    foreach v!=s:
        dist[v] = inf
        pred[v] = null
    dist[s] = 0
    foreach v belonging to V:
        insert(pq, v, dist[v])
    while(!empty(pq)):
        u = del-min(pq)
        foreach edge e = (u,v) belonging to E leaving u:
            if dist[v] > dist[u]+len(e):
                decrease-key(pq, v, dist[u]+len(e))
                dist[v] = dist[u]+len(e)
                pred[v] = e
```

Minimum spanning tree

The greedy algorithm

Red rule

- Let C be a cycle with no red edges
- Select an uncolored edge of C of max cost and color it red

Blue rule

- Let D be a cutset with no blue edges
- Select an uncolored edge in D of min cost and color it blue

Greedy algorithm

- Apply the red and blue rules until all edges are colored. The blue edges form an MST
- Note: can stop once n-1 edges colored blue

Prim's algorithm

```
Initialize S = {s} for any node s, T = []
repeat n-1 times:
    Add to T a min-cost edge with exactly one endpoint in S
    Add the other endpoint to S
```

Kruskal's algorithm

```
Kruskal(V,E,c):
    sort m edges by cost and renumber so that c(e1)<=c(e2)<= ... <= c(em)
    T = []
    foreach v belonging to V:
        make-set(v)
    for i=1 to m:
        (u,v) = ei
        if find-set(u) != find-set(v):
            T = T + {ei}
            union(u,v)
    return T
```

Boruvka's algorithm

Repeat until only one tree.

- Apply blue rule to cutset corresponding to each blue tree
- Color all selected edges blue.

Single-link clustering

k-clustering. Divide objects into k non-empty groups

Goal. Given an integer k, find a k-clustering of maximum spacing.

Min-cost arborescence

Def. Given a digraph G = (V, E) and a root r ∈ V, an **arborescence** (rooted at r) is a subgraph T = (V, F) such that

- T is a spanning tree of G if we ignore the direction of edges.
- There is a (unique) directed path in T from r to each other node v ∈ V.

```
Edmonds-branching(G, r, c):
    foreach v!=r:
        y(v) = min cost of any edge entering v
        c'(u,v) = c'(u,v) - y(v) for each edge (u,v) entering v
    foreach v!=r:
        choose one 0-cost edge entering v and let F* be the resulting set of
edges
    if(F* forms an arborescence):
        return T=(V,F*)
    else:
        C = directed cycle in F*
        contract C to a single supernode, yielding G' = (V', E')
        T' = Edmonds-braching(G', r, c')
        Extend T' to an arborescence T in G by adding all but one edge of C
        return T
```

5. Divide and Conquer

Merge sort

```
Merge-Sort(L):
    if(list L has one element):
        return L
    divide the list into two halves A and B
    A = Merge-Sort(A)
    B = Merge-Sort(B)
    L = Merge(A,B)
    return L
```

Counting inversions

```
Sort-And-Count(L):
    if (list L has one element):
        return (0,L)
    Divide the list into two halves A and B
    (ra, A) = Sort-And-Count(A)
    (rb, B) = Sort-And-Count(B)
    (rab, L) = Merge-And-Count(A,B)
    return (ra+rb+rab, L)
```

Randomized quicksort

Goal. Given an array A and pivot element p, partition array so that:

- Smaller element in left subarray L
- Equal elements in middle subarray M
- Larger elements right subarray R

```
Randomized-Quicksort(A):
    if (array A has zero or one element):
        return
    pick pivot p belonging to A uniformly at random
    (L,M,R) = partition-3-way(A,p)
    Randomized-Quicksort(L)
    Randomized-Quicksort(R).
```

Median and selection

```
Quick-Select(A, k):
    pick pivot p belonging to A uniformly at random
    (L,M,R) = Partition-3-way(A,p)
    if (k<=|L|):
        return Quick-Select(L,k)
    elif (k > |L|+|M|):
        return Quick-Select(R,k-|L|-|M|)
    elif (k==|L|):
        return p
```

Median-of-median selection algorithm

```
MOM-select(A, k):
```

```
    n = |A|
    if (n<50):
        return k-th smallest of element of A via mergesort
    group A into n/5 groups of 5 elements each
    B = median of each group of 5
    p = MOM-select(B, n/10)

    (L,M,R) = partition-3-way(A,p)
    if (k<=|L|):
        return MOM-select(L,k)
    elif (k>|L|+|M|):
        return MOM-select(R, k-|L|-|M|)
    else:
        return p
```

Master theorem

Master theorem. Let $a \geq 1$, $b \geq 2$, and $c \geq 0$ and suppose that $T(n)$ is a function on the non-negative integers that satisfies the recurrence

$$T(n) \;=\; aT\left(\frac{n}{b}\right) \;+\; \Theta(n^c)$$

with $T(0) = 0$ and $T(1) = \Theta(1)$, where n/b means either $\lfloor n/b \rfloor$ or $\lceil n/b \rceil$. Then,

Case 1. If $c > \log_b a$, then $T(n) = \Theta(n^c)$.
Case 2. If $c = \log_b a$, then $T(n) = \Theta(n^c \log n)$.
Case 3. If $c < \log_b a$, then $T(n) = \Theta(n^{\log_b a})$.


Integer multiplication

```
Multiply(x,y,n):
    if (n==1):
        return x * y
    else:
        m = n/2
        a = x/2**m, b = x % 2**m
        c = y/2**m, d = y % 2**m
        e = Multiply(a,c,m)
        f = Multiply(b,d,m)
        g = Multiply(b,c,m)
        h = Multiply(a,d,m)

        return 2^(2m) * e + 2^m * (g+h) + f
```

Matrix multiplication

```
strassen(n,A,B):
    if(n==1):
        return A * B
    partition A and B into sqrt(n)-by-sqrt(n) blocks
    p1 = strassen(n/2, A11,(B11-B22))
    p2 = strassen(n/2, (A11+A12),B22)
    p3 = strassen(n/2, (A21+A22),B11)
```

```
    p4 = strassen(n/2, A22,(B21-B11))
    p5 = strassen(n/2, (A11+A22),(B11+B22))
    p6 = strassen(n/2, (A12-A22),(B21+B22))
    p7 = strassen(n/2, (A11-A21),(B11+B12))

    c11 = p5+p4-p2+p6
    c12 = p1+p2
    c21 = p3+p4
    c22 = p1+p5-p3-p7
    return c
```

Convolution and FFT

```
inverse-FFT(n,y0,y1,...,yn-1):
    if (n==1):
        return y0
    (e0,e1,...,en/2-1) = inverse-FFT(n/2, y0,y2,...,yn-2)
    (d0,d1,...,dn/2-1) = inverse-FFT(n/2, y1,y3,...,yn-1)
    for k = 0 to n/2-1:
        wk = exp(-2*pi*i*k/n)
        ak = ek + w**k * dk
        ak+n/2 = ek - w**k * dk
    return (a0,a1,...,an-1)
```

6.Dynamic Programming

Weight interval scheduling

Greedy method: Earliest finish-time first

```
pj = largest index i<j such that job i is compatible with j.
find-solution(j):
    if (j==0):
        return []
    elif (wi + M[p[j]] > M[j-1]):
        return {j} + find-solution(p[j])
    else:
        return find-solution(j-1)
```

Segmented least squares

OPT(j) = minimum cost for points p1,p2,..., pn

eij = SSE for points pi, pi+1, ..., pj

To compute OPT(j):

- Last segment uses points pi,..., pj for some i<=j
- Cost = eij + c + OPT(i-1)

Bellman equation:

- OPT(j) = 0 if j=0
- OPT(j) = min $_{1<=i<=j}$ {eij + c + OPT(i-1)} if j>0

```
segment-least-squares(n,p1,p2,...,pn,c):
    for j=1 to n:
        for i=1 to j:
            compute the SSE eij for points pi, ..., pj
    M[0] = 0
    for j=1 to n:
        M[j] = min{eij + c + M[i-1]}
    return M[n]
```

Knapsack problem

```
knapsack(n,W,w1,w1,...,wn, v1,v2,...,vn):
    for w=0 to W:
        M[0,w] = 0
    for i=1 to n:
        for w=0 to W:
            if(wi > w):
                M[i,w] = M[i-1,w]
            else:
                M[i,w] = max(M[i-1,w], vi+M[i-1,w-wi])
    return M[n,W]
```

Coin Changing

Given n coin denominations {d1,d2,...,dn} and a target value V.

Def. OPT(v) = min number of coins to make change for v

Bellman equation

- OPT(v) = 0, if v=0
- OPT(v) = $\min_{1<=i<=n}$ {1 + OPT(v-di)},   if v>0

RNA secondary structure

```
rna-secondary-structure(n,b1,b2,...,bn):
    for k=5 to n-1:
        for i=1 to n-k:
            j = i+k
            compute M[i,j] using formula
    return M[1,n]
```

Sequence alignment

Goal. Given two string x1,..., xm and y1,..., yn, find a min-cost alignment

Def. An **alignment** M is a set of ordered pairs xi-yj such that each character appears in at most one pair and no crossings.

Def. The **cost** of an alignment M is:

$$\text{cost}(M) = \underbrace{\sum_{(x_i, y_j) \in M} \alpha_{x_i y_j}}_{\text{mismatch}} + \underbrace{\sum_{i : x_i \text{ unmatched}} \delta + \sum_{j : y_j \text{ unmatched}} \delta}_{\text{gap}}$$

```
sequence-alignment(m,n,x1,...,xm,y1,...,yn,theta,alpha):
    for i=0 to m:
        M[i,0] = i*thera
    for j=0 to n:
        M[0,j] = j*theta
    for i=1 to m:
        for j=1 to m:
            M[i,j] = min(alpha[xi][yj] + M[i-1,j-1], theta+M[i-1,j],
theta+M[i,j-1])
    return M[m,n]
```

Hirschberg's algorithm

Bellman-Ford-Moore algorithm

Lemma. If G has no negative cycle, then there exists a shortest v->t path that is simple(and has <= n-1 edges)

Def. OPT(i,v) = length of shortest v->t path that uses <= i edges

Goal. OPT(n-1, v) for each v

Bellman equation.

- OPT(i,v) = 0,   if i =0 and v = t
- OPT(i,v) = inf,   if i = 0 and v!=t
- OPT(i,v) = min(OPT(i-1,v), min{OPT(i-1,w) + len(v,w)}),   if i > 0

```
shortest-paths(V,E,len,t):
    foreach node v belonging to V:
        M[0,v] = inf
    M[0,t] = 0
    for i = 1 to n-1:
        foreach node v belonging to V:
            M[i,v] = M[i-1,v]
            foreach edge(v,w) belonging to E:
                M[i,v] = min{M[i,v], M[i-1,w] + len(v,w)}
```

Space optimization.

Maintain two 1D arrays

- d[v] = length of a shortest v->t path that we have found so far
- successor[v] = next node on a v->t path

```
Bellman-Ford-Moore(V,E,c,t):
    foreach node v belonging to V:
        d[v] = inf
        successor[v] = null
    d[t] = 0
    for i = 1 to n-1:
        foreach node w belonging to V:
            if(d[v] > d[w] + len(v,w)):
                d[v] = d[w] + len(v,w)
                successor[v] = w
    if (no d[] value changed in pass i):
        stop
```

Distance-vector protocols

Communication network

- Node = router
- Edge = direct communication link
- Length of edge = latency of link

Negative cycles

Negative cycle detection problem

Given a digraph G = (V,E), with edge length len(v,w), find a negative cycle(if one exists)

- Run Bell-Ford-Moore on G' for n'=n+1 passes (instead of n'-1)
- If no d[v] values updated in pass n', then no negative cycles
- Otherwise, suppose d[s] updated in pass n'
- Define pass(v) = last pass in which d[v] was updated
- Observe pass(s) = n' and pass(successor[v]) >= pass(v) - 1 for each v
- Following successor pointers, we must eventually repeat a node.
- The corresponding cycle is a negative cycle`

7.Network Flow

max-flow and min-cut problems

Def. A cut's **capacity** is the sum of the capacities of the edges from A to B

cap(A,B) = $\text{sum}_{e \text{ out of } A}$ c(e)

Def. The **value** of a flow f is: val(f) = $\text{sum}_{e \text{ out of } s}$ f(e) - $\text{sum}_{e \text{ in to } s}$ f(e)

Ford-Fulkerson algorithm

Def. An augmenting path is a sample s->t path in the residual network Gf

```
Augment(f,c,P):
    theta = bottleneck capacity of augment path P
    foreach edge e belonging to P:
        if (e blongs to E):
            f(e) = f(e) + theta
            else:
                f(e_reverse) = f(e_reverse) - theta
    return f
```

```
ford-fulkerson(G):
    foreach edge e belonging to E:
        f(e) = 0
    while(there exists an s->t path P in Gf):
        f = Augment(f,c,P)
        update Gf
    return f
```

max-flow min-cut theorem

val(f) <= cap(A,B)

if val(f) = cap(A,B): then f is a max flow and (A,B) is a min cut

Augmenting path theorem. A flow f is a max flow `iff` no augmenting paths.

capacity-scaling algorithm

Goal. Choose augmenting paths so that:

- Can find augmenting paths efficiently
- Few iterations

```
capacity-scaling(G):
    foreach edge e belonging to E:
        f(e) = 0
    delta = largest power of 2 <= C
    while(delta>=1):
        Gf(delta) = delta-residual network of G with respect to flow f
        while(there exists an s->t path P in Gf(delta)):
            f = Augment(f,c,P)
            update Gf(delta)
        delta = delta/2
    return f
```

shortest augmenting paths

Q: How to choose next augmenting path in Ford-Fulkerson?

A: Pick one that uses the fewest edges.

```
short-augmenting-path(G):
    foreach e belonging to E:
        f(e) = 0
        Gf = residual network of G with respect to flow f
        while(there exists an s->t path in Gf):
            P = BFS(Gf)
            f = Augment(f,c,P)
            update Gf
    return f
```

Dinitz algorithm

Two types of augmentations:

- Normal: length of shortest path does not change
- Special: length of shortest path strictly increases

```
initialize(G,f):
    LG = level-graph of Gf
    P = []
    GOTO advance(s)

advance(v):
    if(v==t):
        augment(P)
        remove saturated edges from LG
        P = []
        GOTO advance(s)

repeat(v):
    if(v==s):
        stop
    else:
        delete v(and all incident edges) from LG
        remove last edge(u,v) from P
        GOTO advance(u)
```

simple unit-capacity network

Def. A flow network is a **simple unit-capacity network** if:

- Every edge has capacity 1
- Every node(other than s or t) has exactly one entering edge, or exactly one leaving edge, or both

Bipartite matching

Given a bipartite graph G = (L ∪ R, E), find a max-cardinality matching

Formulation

- Create digraph G' = {L∪R∪{s,t}, E'}
- Direct all edges from L to R, and assign infinite(or unit) capacity
- Add unit-capacity edges from s to each node in L
- Add unit-capacity edges from each node in R to t

Disjoint paths

Edge-disjoint paths problem

Given a digraph G=(V,E) and two nodes s and t, find the max number of edge-disjoint s->t paths

Menger's theorem The max number of edge-disjoint s->t paths equals the min number of edges whose removal disconnects t from s.

Max-flow formulation. Assign unit capacity to every edge

Edge-disjoint paths problem in undirected graphs

Given a graph G=(V,E) and two nodes s and t, find the max number of edge-disjoint s-t paths

Max-flow formulation. Replace each edge with two antiparallel edges and assign unit capacity to every edge

Theorem. Max number of edge-disjoint s->t paths = value of max flow.

Extensions to max flow

Multiple sources and sinks: max-flow formulation

- Add a new source node s and sink node t
- For each original source node si, add edge(s,si) with capacity inf
- For each original sink node tj, add edge(tj, t) with capacity inf

Circulation with supplies and demonds

Def Given a digraph G = (V, E) with edge capacities c(e) ≥ 0 and node demands d(v), a circulation is a function f(e) that satisfies:

- For each e belonging to E: 0 <= f(e) <= c(e)
- For each v belonging to V: sum_{in} f(e) - sum_{out} f(e) = d(v)
- Add new source s and sink t
- For each v with d(v) < 0, add edge (s, v) with capacity −d(v)
- For each v with d(v) > 0, add edge (v, t) with capacity d(v)

Claim:

G has a circulation **iff** G' has a max flow of value D = $\text{sum}_{positive}$ d(v) = $\text{sum}_{negative}$-d(v)

Survey design

- Design survey asking n1 consumers about n2 products.
- Can survey consumer i about product j only if they own it.
- Ask consumer i between ci and ci' questions.
- Ask between pj and pj' consumers about product j.

Goal. Design a survey that meets these specs, if possible

Max-flow formulation. Model as a circulation problem with lower bounds.

- Add edge (i, j) if consumer j owns product i.
- Add edge from s to consumer j.
- Add edge from product i to t.
- Add edge from t to s.
- All demands = 0.

- Integer circulation $\iff$ feasible survey design

Airline scheduling

- Manage flight crews by reusing them over multiple flights.
- Input: set of k flights for a given day.
- Flight i leaves origin oi at time si and arrives at destination di at time fi.
- Minimize number of flight crews.

Circulation formulation. [to see if c crews suffice]

- For each flight i, include two nodes ui and vi.
- Add source s with demand −c, and edges (s, ui) with capacity 1.
- Add sink t with demand c, and edges (vi, t) with capacity 1.
- For each i, add edge (ui, vi) with lower bound and capacity 1.
- if flight j reachable from i, add edge (vi, uj) with capacity 1

Image segmentation

Projection selection

Projects with prerequisites.

- Set of possible projects P: project v has associated revenue p_v.
- Set of prerequisites E: (v, w) ∈E means w is a prerequisite for v.
- A subset of projects A⊆P is feasible if the prerequisite of every project in A also belongs to A.

Project selection problem. Given a set of projects P and prerequisites E, choose a feasible subset of projects to maximize revenue.

Min-cut formulation

- Assign a capacity of ∞ to each prerequisite edge.
- Add edge (s, v) with capacity pv if pv > 0.
- Add edge (v, t) with capacity −pv if pv < 0.
- For notational convenience, define ps= pt = 0.

Claim. (A, B) is min cut `iff` A− {s} is an optimal set of projects.

Assignment problem

Input. Weighted, complete bipartite graph G = (X∪Y, E) with |X| = |Y|.

Goal. Find a perfect matching of min weight.

Bipartite matching. Can solve via reduction to maximum flow.

Input-queued switching

Input-queued switch

- n input ports and n output ports in an n-by-n crossbar layout.
- At most one cell can depart an input at a time.
- At most one cell can arrive at an output at a time.
- Cell arrives at input x and must be routed to output y.

8. Intractability

Poly-time reductions

A working definition. Those with poly-time algorithms.

yes	probably no
shortest path	longest path
min cut	max cut
2-satisfiability	3-satisfiability
planar 4-colorability	planar 3-colorability
bipartite vertex cover	vertex cover
matching	3d-matching
primality testing	factoring
linear programming	integer linear programming

Reduction. Problem X *polynomial-time reduces to* problem Y if arbitrary instances of problem X can be solved using:

- Polynomial number of standard computational steps, plus
- Polynomial number of calls to oracle that solves problem Y

Notation. X $<=_p$ Y

If x reduces to y, then x is easier than y.

Establish equivalence. If both x $<=_p$ y and y $<=_p$ x, we use notation x $=_p$ y.

Packing and covering problems

Independent-Set. Given a graph G = (V,E) and an integer k, is there a subset of k (or more) vertices such that no two are adjacent?

Vertex-Cover. Given a graph G = (V, E) and an integer k, is there a subset of k (or fewer) vertices such that each edge is incident to at least one vertex in the subset?

Theorem. INDEPENDENT-SET$\equiv_p$ VERTEX-COVER.

Set-Cover. Given a set U of elements, a collection S of subsets of U, and an integer k, are there ≤k of these subsets whose union is equal to U ?

Theorem. Vertex-Cover $<=_p$ Set-Cover

Lemma. G = (V, E) contains a vertex cover of size k `iff` (U, S, k) contains a set cover of size k.

Constraint satisfaction problems

Satisfiability

Literal. A Boolean variable or its negation

Clause. A disjunction of literals.

Conjunctive normal form (CNF). A propositional formula Φ that is a conjunction of clauses.

SAT. Given a CNF formula Φ, does it have a satisfying truth assignment?

3-SAT. SAT where each clause contains exactly 3 literals (and each literal corresponds to a different variable).

Theorem. 3-SAT $<=_p$ independent-set

Construction.

- G contains 3 nodes for each clause, one for each literal.
- Connect 3 literals in a clause in a triangle.
- Connect literal to each of its negations.

Lemma. Φ is satisfiable `iff` G contains an independent set of size k = |Φ|

Basic reduction strategies

- Simple equivalence: INDEPENDENT-SET $\equiv_P$ VERTEX-COVER.
- Special case to general case: VERTEX-COVER $\leq_p$ SET-COVER.
- Encoding with gadgets: 3-SAT $\leq_P$ INDEPENDENT-SET.

Decision problem. Does there exist a vertex cover of size <= k?

Search problem. Find a vertex cover of size <= k

Optimization problem. Find a vertex cover of minimum size

Vertex-Cover $=_p$ Find-Vertex-Cover $=_p$ Find-Min-Vertex-Cover

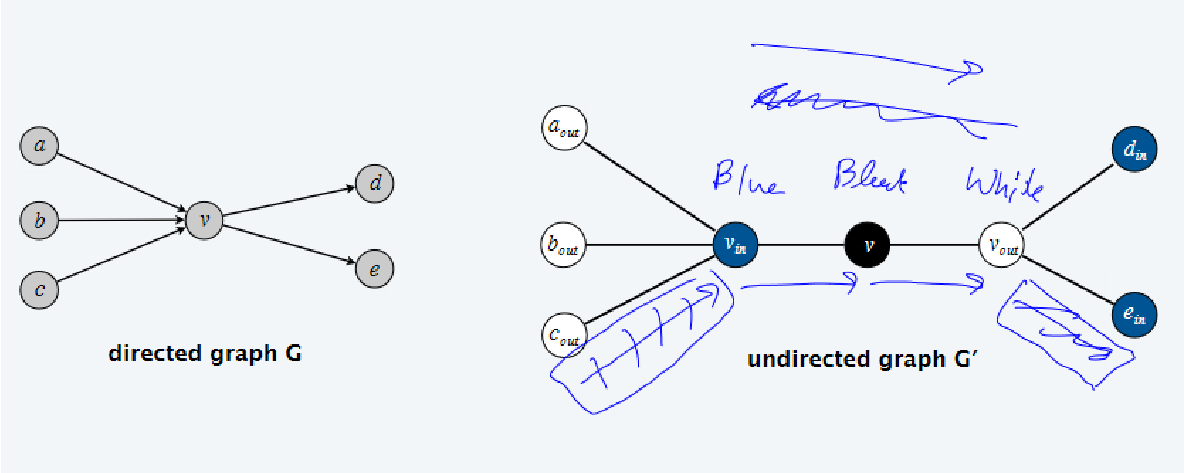
Sequencing problems

HAMILTON-CYCLE. Given an undirected graph G = (V, E), does there exist a cycle Γ that visits every node exactly once?

DIRECTED-HAMILTON-CYCLE. Given a directed graph G = (V, E), does there exist a directed cycle Γ that visits every node exactly once?

Theorem. DIRECTED-HAMILTON-CYCLE$\leq_P$ HAMILTON-CYCLE.

Pf.

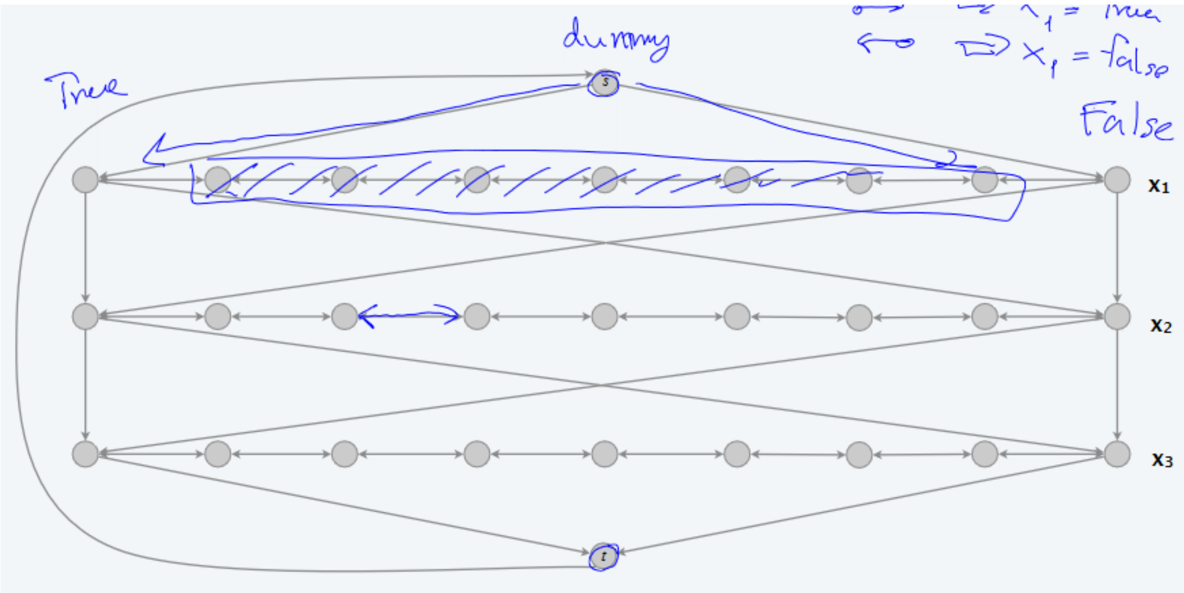


Lemma. G has a directed Hamilton cycle `iff` G' has a Hamilton cycle

Theorem. 3-SAT ≤P DIRECTED-HAMILTON-CYCLE.

Construction. Given 3-SAT instance Φ with n variables xi and k clauses.

- Construct G to have 2^n Hamilton cycles.
- Intuition: traverse path i from left to right ⇔ set variable xi= true



Construction. Given 3-SAT instance Φ with n variables xi and k clauses.

- For each clause: add a node and 2 edges per literal.

Lemma. Φ is satisfiable `iff` G has a Hamilton cycle.

Partitioning problems

3-dimensional matching

3-D-Matching. Given 3 disjoint sets X, Y, and Z, each of size n and a set T ⊆X×Y×Z of triples, does there exist a set of n triples in T such that each element of X∪Y∪Z is in exactly one of these triples?

Theorem. 3-SAT ≤P 3D-MATCHING

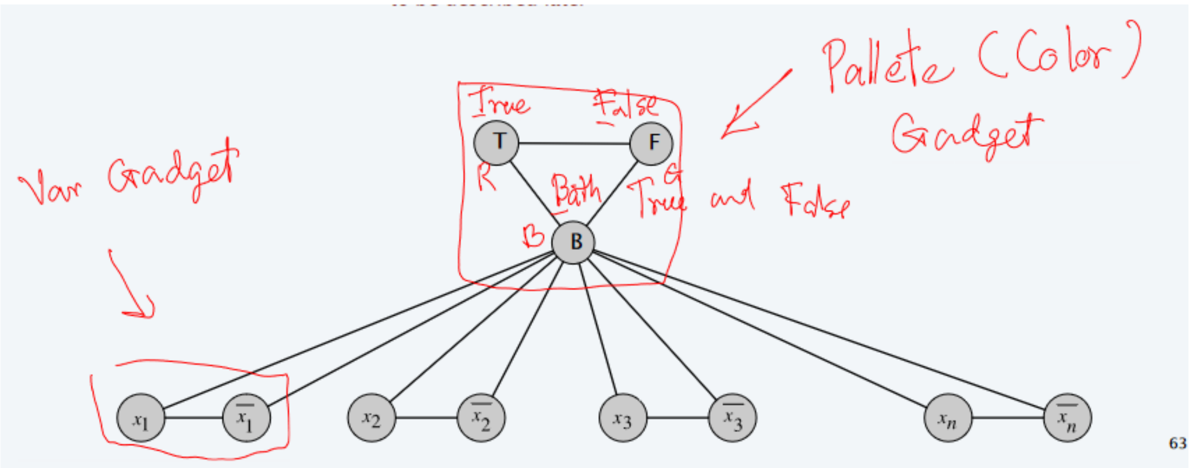
Graph coloring

3-COLOR. Given an undirected graph G, can the nodes be colored black, white, and blue so that no adjacent nodes have the same color?

Theorem. 3-SAT≤P 3-COLOR.

Construction

- Create a graph G with a node for each literal.
- Connect each literal to its negation.
- Create 3 new nodes T, F, and B; connect them in a triangle.
- Connect each literal to B.
- For each clause Cj, add a gadget of 6 nodes and 13 edges.



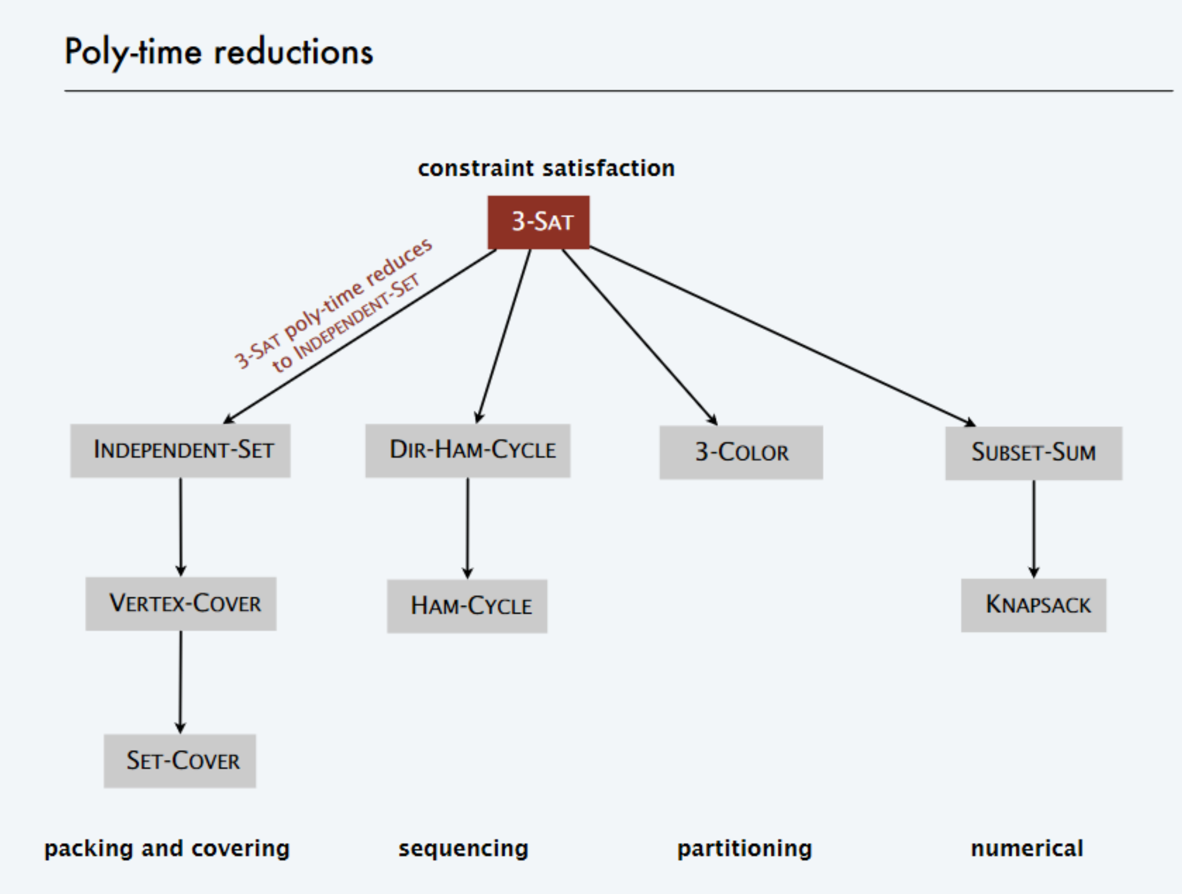
Numerical problems

Subset sum

Subset sum. Given n natural numbers w1, ..., wn and an integer W, is there a subset that adds up to exactly W?

Theorem. 3-SAT≤P SUBSET-SUM.

Poly-time reduction



P vs. NP

P

Decision problem

- Problem X is a set of strings
- Instance s is one string
- Algorithm A solves problem X: A(s) = yes if s belongs to X; no if s doesn't belong to X

Def. Algorithm A runs in **polynomial time** if for every string s, A(s) terminates in ≤p(|s|) "steps," where p(·) is some polynomial function.

Def. P = set of decision problems for which there exists a poly-time algorithm

NP

Def. Algorithm C(s, t) is a **certifier** for problem X if for every string s : s∈X `iff` there exists a string t such that C(s, t) = yes.

Def. NP = set of decision problems for which there exists a poly-time certifier.

- C(s, t) is a poly-time algorithm.
- Certificate t is of polynomial size: |t|≤p(|s|) for some polynomial p(·)

P. Decision problems for which there exists a poly-time algorithm.

NP. Decision problems for which there exists a poly-time certifier.

EXP. Decision problems for which there exists an exponential-time algorithm.

NP-complete

NP-complete. A problem Y∈NP with the property that for every problem X∈NP, $X \leq_P Y$.

Theorem. SAT ∈ NP-complete

Recipe. To prove that Y∈NP-complete:

- Step 1. Show that Y∈NP.
- Step 2. Choose an NP-complete problem X.
- Step 3. Prove that $X \leq_P Y$.

Proposition. If X∈NP-complete, Y∈NP, and $X \leq_P Y$, then Y∈NP-complete.

co-NP

Def. Given a decision problem X, its complement not(X) is the same problem with the yes and no answers reversed.

co-NP. Complements of decision problems in NP.

Theorem. If NP ≠ co-NP, then P ≠ NP.

NP-hard

NP-hard. A problem such that every problem in NP poly-time reduces to it

Summarize

- P: 多项式时间内可解的问题
- NP: 多项式时间内可验证的问题
- NPC: 所有NP问题都可以归约到这个NP问题
- NP-Hard: 所有NP问题都可以归约到这个问题