Input size: integer n: $\log n$

n numbers: n

$G=(V,E)$: $n+m$ $\boxed{m=|E|\ n=|V|}$

$g(0) = 1$ and $g(1) = 2$. Fill in the missing parts in lines 2 and 5 of the following algorithm for computing $g(n)$:

input n;
$x \leftarrow$; $y \leftarrow$;
for $i = 1$ **to** n **do**
  $y \leftarrow 2y + x$;
  $x \leftarrow \frac{y-x}{2}$;
end for;
output y.

$g(-1)$	$g(0)$	$g(1)$	$g(2)$	$g(3)$
	1	2	5	
	x	y		
	x	y		

$i=1$, $y=g(1)$ Base case. $n=1$ $2y+x=2$; $n=0$ $y=1$

$x\in 0$, $y\in 1$

Divide & Conquer

Master Theorem

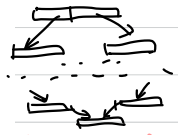
$T(n) = aT(\frac{n}{b}) + O(n^d)$ $a\geq 1,\ b>1,\ d\geq 0$

$$T(n)=\begin{cases} O(n^{\log_b a}) & \log_b a > d \\ O(n^d \log n) & \log_b a = d \\ O(n^d) & \log_b a < d \end{cases}$$

1° $T(n)=2T(\frac{n}{2})+\Theta(n\log n) \Rightarrow \Theta(n\log^2 n)$

2° $T(n)=T(\frac{n}{3})+T(\frac{2n}{3})+\Theta(n) \Rightarrow \Theta(n\log n)$

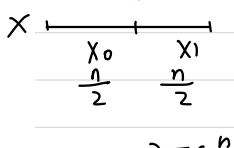
Merge sort $O(n\log n)$



$T(n)=2T(\frac{n}{2})+O(n)$

$T(n)=O(n\log n)$

Multiplication $O(n^{1.59})$



$x\cdot y=(x_0\cdot 10^{\frac{n}{2}}+x_1)\cdot(y_0\cdot 10^{\frac{n}{2}}+y_1)$

$= x_0y_0 10^n + (x_1y_0+x_0y_1)10^{\frac{n}{2}} + x_1y_1$

$= x_0y_0 10^n + \left((x_0+x_1)(y_0+y_1)-x_0y_0-x_1y_1\right)\cdot 10^{\frac{n}{2}} + x_1y_1$

$T(n)=3T(\frac{n}{2})+O(n)$ $T(n)=O(n^{\log_2 3})$

Matrix Multiplication $O(n^{2.81})$

We need only 7 multiplications to compute the product of two 2×2 matrices.

$$\begin{pmatrix} a & b \\ c & d \end{pmatrix}\begin{pmatrix} e & f \\ g & h \end{pmatrix} = \begin{pmatrix} p_5+p_4-p_2+p_6 & p_1+p_2 \\ p_3+p_4 & p_1+p_5-p_3-p_7 \end{pmatrix},$$

where $p_1=a(f-h), p_2=(a+b)h, p_3=(c+d)e, p_4=d(g-e), p_5=(a+d)(e+h), p_6=(b-d)(g+h)$ and $p_7=(a-c)(e+f)$.

$T(n)=7T(\frac{n}{2})+O(n^2)$ $T(n)=O(n^{2.81})$

Fibonacci

1. Use array

Fibonacci(n): compute Fibonacci number f_n.
$x \leftarrow 1$; $y \leftarrow 0$;
for $i = 1$ **to** n **do**
  $y \leftarrow x + y$;
  $x \leftarrow y - x$;
end for;
print y.

$O(n^2)$.
$O(n)$ operations
each operation $O(i)$ input size

2. Divide & Conquer $O(n^{1.59})$

$$\begin{pmatrix} f_n \\ f_{n+1} \end{pmatrix} = \begin{pmatrix} 0 & 1 \\ 1 & 1 \end{pmatrix}^n \begin{pmatrix} 0 \\ 1 \end{pmatrix}$$

3. DP Avoid repeating calculating some num calculated before

4. NPC. Input len $=\log n$, Output len $\geq \log(1.68^n)$, exponential.

Walk Edges & vertices can repeate

Open walk End & start vertex different

Closed walk End & start same vertex

Tour / Trail No repeated edges, can repeate vertex

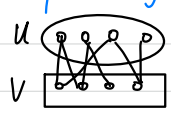
Circuit Close trail

Eulerian trail Visit every edge once without repeat

Path No repeated edges or vertices, open

Cycle No repeated edges or vertices, close

Bipartite graph / Bigraph



vertices divided into two independent sets.
edges connect one in U to one in V.

Euler cycle / circuit / tour close walk, each edge once
iff every vertex has an even degree; digraph: in=out

7. *Block*: A *block* B in a graph is a maximal subgraph that is connected and has no cut vertex. Here *maximal* means that any supergraph of B is either disconnected or has a cut vertex. Blocks can also be defined as equivalence classes of binary relation R such that for any two edges e and e', eRe' if either $e=e'$ or there is a cycle containing both e and e'.

   We can use DFS to find all cut vertices of G, and then use them together with the DFS tree from bottom-up to partition edges of G into blocks. Note that an edge is a bridge iff it forms a block by itself.

Systematic Search

- BFS (least recently visited frontier, queue $\Rightarrow O(m+n)$
- DFS most recently visited frontier, stack

BFS

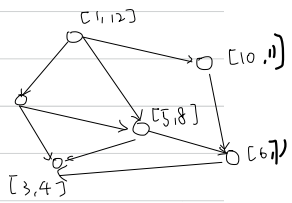
Shortest Path BFS visit in order $N^1(s), N^2(s), \dots$

Every (s,v)-path in BFS = shortest $(s-v)$ path in G

DFS

Timestamping

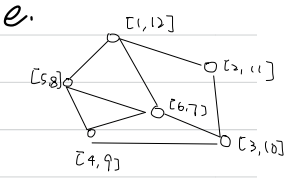
Tree edge (in forest)	$[u\,[v\,]_v]_u$
Forward edge (to descendants)	$[u[v\,]_v]_u$
Back edge (to ancestor)	$[v\,[u\,]_u]_v$
Cross edge (all other)	$[v]_v\ \ [u]_u$



If every DFS has back edge, there's cycle.

Eulerian tour (every v, even degree)

Algo1. Merging closed trails.



Algo2. Avoiding bridges, choose edge not a bridge

Topological Sort DAG. edges v_iv_j, $i<j$

DAG has a leat one source, ≥ 1 sink.

Algo.1. Repeatedlly delete source

find source → delete source → other edge in-deg −1 → if in-deg $=0$ → source → delet...

Algo 2. $O(n)$ radix/bucket sort post, $1\leq post(v)\leq 2n$

p.s. smallest post(v) is sink.

Cut vertex A vertex, delete it → disconnect G

Bridge An edge, delete it → disconnect G
↳ iff not in a cycle

DFS find cut vertex & bridge in $O(m+n)$

In a DFS tree T, (a) the root is a cut vertex of G iff it has at least two children, and (b) a non-root vertex v is a cut vertex of G iff it has a child v' none of whose descendants has a back edge to a proper ancestor of v.

For case (b), the deletion of v will cut the DFS tree into several pieces, and in order for these pieces to be connected in G, at least one vertex in each piece has a back edge connected to a proper ancestor of v.

For a vertex v, define $\alpha(v)$ to be the pre number of the oldest ancestor of v that can be reached through a back edge from a descendant of v. Set $\alpha(v)=pre(v)$ if no such back edge.

Fact: A non-root vertex v is a cut vertex iff v has a child v' such that $\alpha(v')\geq pre(v)$.

Note that $\alpha(v)$ can be computed in linear time in a bottom-up fashion starting from leaves of DFS trees.

8. *Strongly connected component* (scc): Two vertices u and v in a digraph are *strongly-connected* if G contains both (u,v)- and (v,u)-paths. Strongly-connectedness is an equivalence relation and thus partitions vertices into sccs.

9. *Meta-graph*: For every digraph G, we can construct a meta-graph G^* whose nodes are sccs of G and there is an edge from node x to node y iff there is an edge in G from a vertex in the corresponding scc of x to a vertex in the corresponding scc of y. A meta-graph is a dag.

   If there is an edge goes from one scc X to another scc Y, then all edges between X and Y are from X to Y, since otherwise X and Y would be in a single scc.

10. *Sink scc*: An scc is a sink scc if it is a sink in the meta-graph. For any vertex v in a sink scc S, the DFS procedure dfs(v) retrieves exactly all vertices of S. This is because dfs(v) terminates when all vertices reachable from v have been visited.

11. *Highest post number*: The vertex with the highest post number in a DFS of G always belongs to a source scc.

    Consider two sccs X and Y with an edge going from X to Y. If DFS visits X before Y, then DFS will visit all vertices in X and Y before it gets stuck, and the first visited vertex in X has a higher post number than any vertex in Y. If DFS visits Y first, then it will visit all vertices in Y before visiting any vertex in X, and vertices in X have higher post numbers than vertices in Y.

    Therefore the highest post number in X is bigger than the highest post number in Y, implying that the vertex with the highest post number is in a source scc.

    Question: Does every vertex in X has a higher post number than vertices in Y?

    Question: Does the vertex with the lowest post number in a DFS of G always belong to a sink scc?

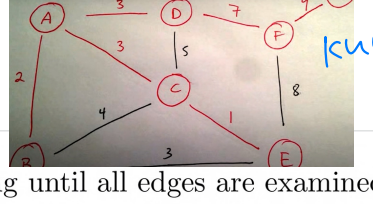
12. *Reverse graph*: The reverse graph G^R of G is obtained from G by reversing the direction of each edge of G. The vertex with the highest post number in a DFS of the reverse graph G^R of G always belongs to a sink scc.

13. *Decomposition algorithm*: Use DFS twice to obtain a linear-time algorithm.

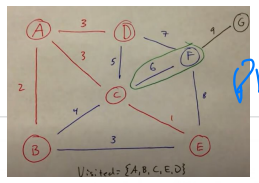
    Step 1 Run a DFS on the reverse graph G^R.

    Step 2 Run a DFS on G following decreasing order of post numbers from the DFS in Step 1 to retrieve sccs.

MST



kuskal



Prim

Repeat the following until all edges are examined:
Choose a maximum-weight edge e among all un-examined edges in G;
If e is not a bridge of G **then** delete it from G.

Repeatedly apply either red or blue rules until all edges are coloured:

(a) *Blue rule*: Choose a cut without blue edge, select an uncoloured edge of minimum weight in the cut, and colour it blue.

(b) *Red rule*: Choose a cycle without red edge, select an uncoloured edge of maximum weight in the cycle, and colour it red.

blue → MST

(a) *Kruskal's algorithm (1956)*
Apply the following step to edges in non-decreasing order by weight:
If the two ends of the current edge lie in two different blue trees, color it b
otherwise color it red.

(b) *Prim's algorithm (1957, Jarník 1930)*
Choose a vertex s as the starting vertex and repeat the following $n-1$ times:
Let T be the blue tree containing s.
Select a minimum-weight edge in the cut $[V(T), V - V(T)]$ and color it blue.

(c) *Boruvka's algorithm (1926)*
Repeat the following until there is a single blue tree:
For every blue tree T, select a minimum-weight edge in the cut $[V(T), V - V($
Color all selected edges blue.
Note: All edge weights need to be distinct in order for Boruvka's algorithm to work correctly, otherwise we need a tie-breaking rule for edges of equal weight.

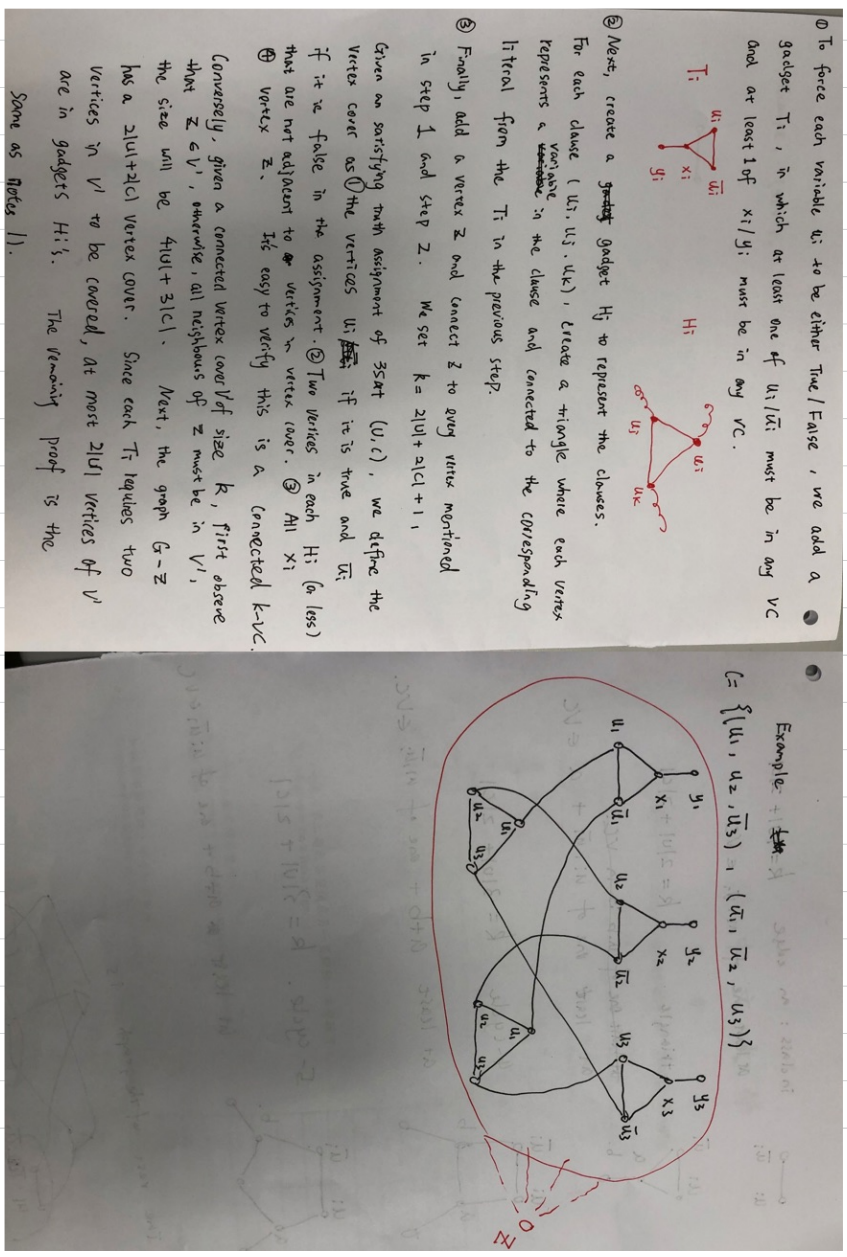
Shortest Path
No negative edge: Dijkstra,
O(m+nlogn), generalization of BFS

Initialize $\delta(s)$ to 0, $p(x) = x$, and $\delta(v)$ to ∞ for all other vertices v;
$S = \emptyset$ and $T = \emptyset$;
for $i = 1$ **to** n **do**
  Select a vertex $x \in V - S$ with smallest $\delta(x)$;
  Add x to S;
  Add edge $\{p(x), x\}$ to T;
  for each edge xy **do**
    Update $\delta(y)$: **if** $\delta(y) > \delta(x) + w(xy)$
      then $\delta(y) = \delta(x) + w(xy)$ and $p(y) = x$.

↳ parent of x in shortest path tree

Has negative edge: Bellman Ford, O(mn)

Repeat the following step $n-1$ times: **for** every edge uv **do** Update(uv).
Update(uv): **if** $\delta(v) > \delta(u) + w(uv)$ **then** $\delta(v) \leftarrow \delta(u) + w(uv)$.



HW Solutions

HW4

(a) For a weighted graph $G = (V, E; w)$ with $w : E \to Z$ and integer W, determine whether G contains a path with total weight at most W.

(b) Determine whether a graph G contains $\geq k$ vertices V' such that for any two vertices $u, v \in V'$ their distance $d_G(u, v) \leq d$.

(a) Set w = -1 for every edge in E and W = -(|V| - 1).

The problem contains Hamiltonian path as a special case.

(b) Set d = 1 and this means every two vertices u,v in V' must be adjacent.

The problem contains CLIQUE as a special case.

(c) Determining whether a graph G contains $\leq k$ vertices whose deletion results in at least l components.

(d) Determining whether it is possible to choose tiles from n rectangular tiles of dimensions $w_i \times h_i$, $1 \leq i \leq n$, to cover exactly a rectangular wall of dimension $W \times H$.

(c) Set L= |V| - k, and since G has |V| vertices, the L connected components must be an independent set.

The problem contains VERTEX COVER as a special case.

(d) Set hi = 0 for all i, and H=0.

The problem contains SUBSET SUM as a special case.

delete at most k vertices to destroy all triangle

- Given an instance (G,k) of VERTEX COVER, we construct an instance (G', k) of TRIANGLE REMOVAL as follows:

1. Create a copy of G.
2. For each edge e=v_iv_j, we add a new vertex u_{ij}, and edges $u_{ij}v_i$ and $u_{ij}v_i$.
3. k remains the same.

Suppose G has a vertex cover X of size k , then removal of X in G' also delete all triangles.

Conversely, suppose deleting Y of size k removes all triangles in G', we can find verte cover Y' in G of size k. For each v in Y and we add it to Y' and for each vertex u in Y, we add a neighbor of u.

Determine G has VC at most $\frac{|V|}{3}$ vertices

- The problem is in NP.
- Given an instance (G,k) of VERTEX COVER, we consider two cases:

1. k>|V|/3 Add 3k-|V| isolated vertices to G and call the new graph G'
   Now G has a k-vertex cover if and only if G' has a |V'|/3 vertex cover.

2. k<|V|/3 Add |V|-3k new edges disjoint to G.

Now G' has 3|V|-6k vertices and note that G' has a vertex cover of size |V|-2k i which |V|-3k are new vertices, therefore G has a vertex cover of size k.

Conversely, if G has a vertex cover X of size k, we can cover G' by adding one endpoint of new edge into X to cover all edges. X' has size |V|-2k= |V'|/3.

determine K-VC. Cover ≥ |E|-2 edges

- Add two new edges ab and cd to G and call the new graph G'.

- Suppose we have a vertex cover X of G (|X|=k and X covers all edges of G), then in G' we see that X covers all but 2 edges in G' (|E'|= |E|+2).

- Conversely, G' has a set Y of k vertices covering |E'|-2 = |E| edges, the two edges may lie in G or G'-G. If there is an edge uv in G not covered by Y, then replace the vertex (a/b/c/d) in Y with u and get Y'. After this replacement Y' is a vertex cover of G of size k.

Greedy

3. *Greedy activity-selection algorithm*: $O(n \log n)$.

Assume $f_1 \leq f_2 \leq \ldots \leq f_n$ (otherwise sort activities w.r.t. finish times). Set A contains a solution.

$A \leftarrow \{a_1\}$;
$j \leftarrow 1$;
for $i = 2$ **to** n **do**
  if $s_i \geq f_i$ **then** $A \leftarrow A \cup \{a_i\}$ and $j \leftarrow i$.

0-1 Knapsack problem: Set S of n items and weight constraint $W \in Z^+$, where item i has value v_i and weight w_i with $v_i, w_i \in Z^+$. The objective is to choose a subset S' of S to maximize its total value subject to the total weight constraint W.

The *Fractional Knapsack* problem is the same as 0-1 Knapsack but we can choose fractions of items.

For Fractional Knapsack, we can always select an item with highest v_i/w_i and pack as much as possible to obtain an optimal solution, but this greedy method fails to work for the 0-1 knapsack problem. In fact, as we will see later, the latter problem is NP-hard and very unlikely to admit a polynomial-time algorithm.

find a value + recurrence relation + bottom up + book-keeping

Dynamic Programming

- Tabular method. sub-problem solution dependent
- Optimal solution obtain from sub-prob's optimal solution

Shortest path in DAG d(vi): distance from s to vi

$d(v_i) = \min\{ d(v_j) + w(v_j, v_i) \cdot \ v_j v_i \in E \}$

7. *Shortest paths with $\leq k$ edges.* Find a shortest (s,t)-path with at most k edges.

Let $d(v,i)$ be the length of a shortest (s,v)-path with at most i edges. Then

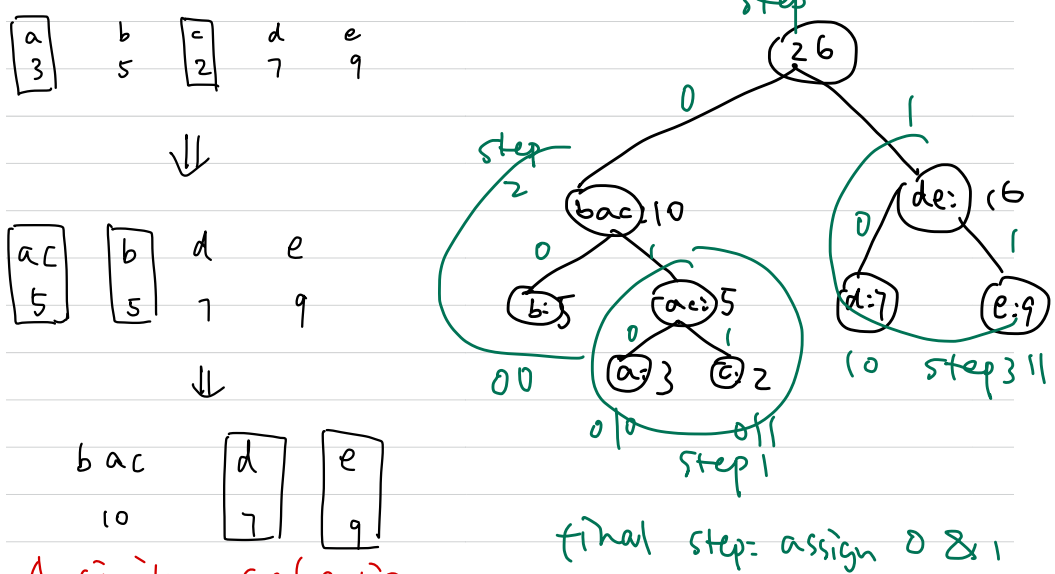
$$d(v,i) = \min_{uv \in E}\{d(v,i-1), d(u,i-1) + w(uv)\}.$$

LIS: Longest Increasing Subsequence L(j): len(longest) end at j

$L(j) = 1 + \max\{L(i) : a_i < a_j\}$

$\text{len}(LIS) = \max_{i \leq j \leq n} L(j)$

Huffman



final step: assign 0 & 1

Activity selection

For interval i, let $r(i)$ be the rightmost interval in $\{1, \cdots, i\}$ that is compatible with i (set $r(i) = 0$ if no such interval). Let $opt(i)$ be the value of an optimal solution for intervals $\{1, \cdots, i\}$. Then

$$opt(i) = \max\{w_i + opt(r(i)), opt(i-1)\}.$$

Interval graph G: Vertex v_i of weight w_i corresponds to interval I_i, and two vertices are adjacent iff their corresponding intervals overlap. The problem is equivalent to the maximum-weight independent set problem on G.

Min-weight VC of tree

9. *Minimum-weight vertex cover in trees*: Find a vertex cover of minimum weight for a weighted tree $T = (V, E, w)$, $w : V \to Z^+$.

Make the input tree T a rooted tree by arbitrarily choosing a vertex as the root r. Denote the children of r by $v_1, \ldots, v_k$ and subtrees rooted at them by $T_1, \ldots, T_k$.

Let $\alpha(T)$ be the weight of a minimum-weight vertex cover of T. In order to compute $\alpha(T)$, we define

$$\beta(T) = \min\{\sum_{v \in V'} w(v) : V' \text{ is a vertex cover of } G \text{ that uses the root of } T\}.$$

with root no root

Then $\beta(T) = w(r) + \sum_{i=1}^{k} \alpha(T_i)$ and $\alpha(T) = \min\{\beta(T), \sum_{i=1}^{k} \beta(T_i)\}$.

Problem: Design an $O(k^2 n)$-time algorithm to find k vertices in a tree to cover the maximum number of edges.

subset sum

$t(i,j) = \begin{cases} s_i \leq j \ \wedge \ t(i-1, j-s_i) & \text{choose } s_i \\ t(i-1, j) & \text{not choose } s_i \end{cases}$

return 0,1

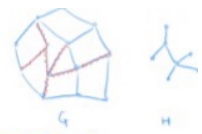
t(i-1, j-si) t(i-1, j) t(i, j)

O(nN) row x col

size(input) = n + log N not polynomial

NPC Restrictions

1. Longest Path



Instance: Graph G and positive integer k.

Question: Does G contain a path of length at least k?

(Proof: Set $k = n - 1$ and we have Hamiltonian Path.) (length = #vertices - 1

2. Dense Induced Subgraph edges among those vertices are all remained

Instance: Graph G, positive integers k and l.

Question: Does G contain an induced subgraph on k vertices that has at least l edges?

(Proof: Set $l = \binom{k}{2}$ and we get Clique.) Independent set not connected fully connected

3. Degree Constrained Spanning Tree

Instance: Graph G and positive integer d.

Question: Does G contain a spanning tree whose maximum vertex degree is at most d?

(Proof: Set $d = 2$ and we get Hamiltonian Path.)

4. Hitting Set

Instance: Collection C of subsets of S, positive integer k.

Question: Does S contain a subset S' with at most k elements such that S' contains at least one element from each subset in C?

(Proof: Set C to be a collection of pairs, and we get Vertex Cover.)

5. Minimum Cover

Instance: Collection C of subsets of S, positive integer k.

Question: Is there a subcollection C' of C with $|C'| \leq k$ such that every element in S appears in at least one member of C'?

(Proof: Set C to be a collection of triples from S and $k = |S|/3$, and we get Exact Cover By 3-Sets.)

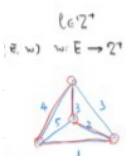
6. Subset Sum

Instance: Set S of positive integers, positive integer W.

Question: Is there a subset S' of S such that $\sum_{x \in S'} x = W$?

(Proof: Set $W = \frac{1}{2}\sum_{x \in S} x$ and we get Partition.)

* Traveling salesman. tour use edge once visit all vertex once, length ≤ L.



V* I: G=(V,E,w), V'⊆V, starting vertex S.
Q: Closed walk, start S, visit all edges in V', end S. length of the walk ≤ L?

TSP. V'=V

Spanning cycle of minimum weight

HC G

l = n

7. Knapsack

Instance: Set U, "size" $s(u) \in Z^+$ and "value" $v(u) \in Z^+$ for each $u \in U$, size constraint $S \in Z^+$ and value goal $V \in Z^+$.

Question: Is there a subset U' of U such that $\sum_{u \in U'} s(u) \leq S$ and $\sum_{u \in U'} v(u) \geq V$? size ≤ S value ≥ V

(Proof: Set $s(u) = v(u)$ for each $u \in U$ and $S = V$, and we get Subset Sum.)

8. Multiprocessor Scheduling

Instance: Set T of "tasks", "length" $l(t) \in Z^+$ for each task $t \in T$, number $m \in Z^+$ of "processors", and "deadline" $d \in Z^+$.

Question: Is there an assignment of all tasks in T to m processors so that all of them can be finished before the deadline d?

(Proof: Set $m = 2$, $d = \frac{1}{2}\sum_{t \in T} l(t)$, and we get Partition.)

9. Edge Packing vertices are limited, try to put edges in

Instance: Graph G, positive integers k and l.

Question: Does G contain k edges that are incident with at most l vertices.

(Proof: Set $k = \binom{l}{2}$ and we get Clique.)

{Initialization}
for $j = 0$ **to** N **do if** $j = 0$ or $j = s_1$ **then** $t(1,j) \leftarrow 1$ **else** $t(1,j) \leftarrow 0$;
for $i = 2$ **to** n **do** $t(i,0) \leftarrow 1$;

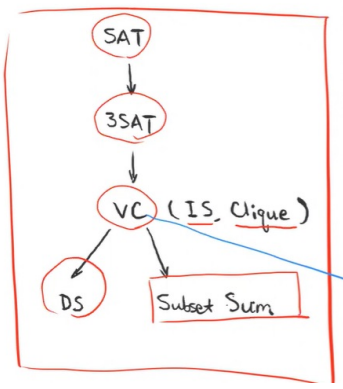
for $i = 2$ **to** n **do**
  for $j = 1$ **to** N **do**
    if $t(i-1,j) = 1$ or $(j \geq s_i$ and $t(i-1, j-s_i) = 1)$
      then $t(i,j) \leftarrow 1$ **else** $t(i,j) \leftarrow 0$.

NP-complete

(a) Prove that $\Pi \in NP$. (Usually very easy!)

(b) Choose an NP-complete problem Π'. (Choosing the right problem Π' is not that easy, but experience helps a lot.)

(c) Prove that $\Pi' \leq_P \Pi$. (Key step whose difficulty varies greatly. DON'T REDUCE Π TO Π'!)



Hamiltonian Cycle

Path

VC $\propto$ 2SAT (weighted)

$C = \{u_1, \bar{u}_2\}\{u_1, u_3\}\{u_2, u_4\}\{u_1, \bar{u}_4\}$

$F_C = (u_1 \vee \bar{u}_2) \wedge (u_1 \vee u_3) \wedge (u_2 \vee u_4) \wedge (u_1 \vee \bar{u}_4)$

$O(m+n)$

$F_G = \bigwedge_{v_i v_j \in E} v_i \vee v_j$

edge $v_i v_j$ is covered

$v_i \vee v_j$

G has a k-vc iff

F_G has a truth assignment of Hamming weight $\leq k$ that satisfies F_G

number of 1's in the assignment.

2SAT $\propto$ VC

$U = \{u_1, u_2, u_3, u_4\}$

$C = \{\{u_1, u_2\}\{u_1, \bar{u}_3\}\{u_2, \bar{u}_4\}\{\bar{u}_3, \bar{u}_4\}\}$

u v

→ Construct graph G from C in polytime s.t. C is satisfiable iff G has a vertex cover of size $\leq k$.

$k = |U|$

Christofides Algo for TSP

Algo: T: MST of G

O: the set of vertices with odd degree in T

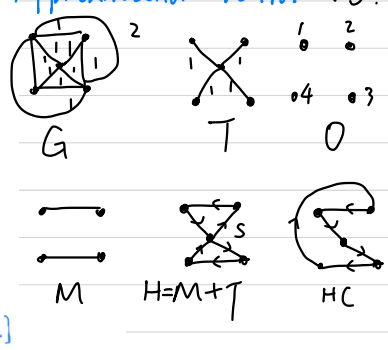
M: Min-weight perfect matching of induced graph of O

H: combination of M and T.

E: Eulerian circuit of H

HC of E by shortcutting

Approximation ratio: 1.5.

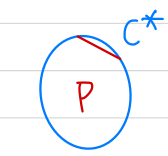


$w(T) < w(C^*\text{-edge}) < w(C^*)$

MST spanning tree $C^* = opt(G)$

$w(M) < w(C^*)/2$

$w(E) = w(M) + w(T) < \frac{3}{2} w(C^*)$



T: MST of G

W: closed walk of T

C: W → HC by shortcuts

$w(C) \leq w(W)$

$w(W) = 2w(T)$

$\Rightarrow w(C) \leq 2w(T)$

C^*: Optimal HC.

P: C^* - one edge get optimal HP.

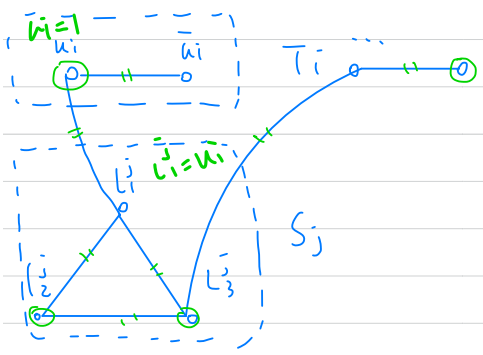
$w(P) \geqslant w(T)$

Path is a spanning tree. T is MST.

$\Rightarrow w(C^*) \geqslant w(T)$

$\Rightarrow w(C) \leq 2w(C^*)$

3SAT $\propto$ VC



$k = |U| + 2|C|$ = #variables + #clauses $\times 2$

VCL

✓ 1. One-Sided Dominating Set

Instance: Bipartite graph $G = (X, Y; E)$ and positive integer k.

Question: Does X contain at most k vertices X' such that every vertex in Y is adjacent to at least one vertex in X'?

Reduction from Vertex Cover by replacing each edge $e = uv$ of a graph $G' = (V', E')$ with two edges ue and ev to form a bipartite graph $G = (X, Y; E)$ with $X = V'$ and $Y = E'$.

✓ 2. Dominating Set

Instance: Graph G and positive integer k.

Question: Does G contain a dominating set with at most k vertices, i.e., a set V' of at most k vertices such that every vertex in $V - V'$ is adjacent to at least one vertex in V'?

Reduction from Vertex Cover. For an arbitrary instance (G, k) of Vertex Cover, we construct an instance (G', k') of Dominating Set by replacing each edge uv in G by the graph H_{uv} in Figure 1 and setting $k' = |E(G)| + k$.

✓ 3. Dag Deletion

Instance: Digraph G and positive integer k.

Question: Can we delete at most k vertices from G to obtain a dag?

Reduction from Vertex Cover by replacing each edge uv with two edges uv and vu to obtain a digraph.

✓ 4. 3-Satisfiability

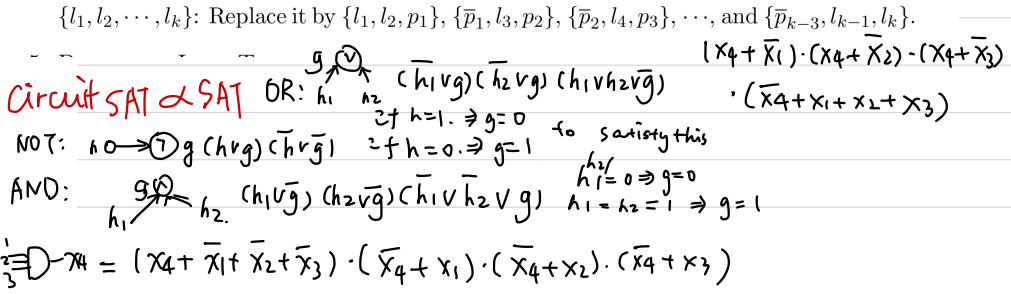
Reduction from Satisfiability. For a clause C, if $|C| \neq 3$ then we replace it by a collection of 3-element clauses as follows:

$\{l\}$: Replace it by $\{l, p, q\}, \{l, \bar{p}, q\}, \{l, p, \bar{q}\}$ and $\{l, \bar{p}, \bar{q}\}$.

$\{l_1, l_2\}$: Replace it by $\{l_1, l_2, p\}$ and $\{l_1, l_2, \bar{p}\}$.

$\{l_1, l_2, \cdots, l_k\}$: Replace it by $\{l_1, l_2, p_1\}, \{\bar{p}_1, l_3, p_2\}, \{\bar{p}_2, l_4, p_3\}, \cdots$, and $\{\bar{p}_{k-3}, l_{k-1}, l_k\}$.

Circuit SAT $\propto$ SAT



Now we improve this to a $\frac{3}{2}$-approximation. Let O be the set of odd-degree vertices of M. The set O contains all leaves of M and perhaps some internal vertices as well. It is also the case that $|O|$ is even, because the sum of the degrees of all the nodes must be even, because it is exactly twice the number of edges.

Thus the complete graph on O has a perfect matching, and we can find a minimum weight perfect matching P in polynomial time. We claim that $d(P) \leq d(T^*)/2$. Let N^* be an optimal TSP tour on just the odd-degree nodes O. Let N_1 and N_2 be the two perfect matchings on O obtained by taking the edges of N^* alternately. Then

$$\begin{aligned} d(P) &\leq \min d(N_1),\ d(N_2) && (4)\\ &\leq d(N^*)/2 && (5)\\ &\leq d(T^*)/2. && (6) \end{aligned}$$

The inequality (4) is from the fact that N_1 and N_2 are perfect matchings on O, and P is a minimum-weight perfect matching on O. The inequality (5) is from the fact that the minimum of $d(N_1)$ and $d(N_2)$ is at most the average. For the inequality (6), get a TSP tour on O from T^* by skipping the even-degree vertices. By the triangle inequality, the length of the resulting tour is no worse than $d(T^*)$, and $d(N^*)$ is no worse than this because it is an optimal tour on O.

Now take the graph with edges E consisting of the edges P and M. This graph may contain multiple edges, since P and M may intersect. But this is ok—if an edge is in both P and M, we just count it twice. Two facts to notice are:

- All nodes have even degree, because we added just one edge of P incident to each node of O.
- $d(E) \leq 3d(T^*)/2$, because $d(M) \leq d(T^*)$ and $d(P) \leq d(T^*)/2$.

Now because each node is of even degree, we can find an Eulerian tour, a tour traversing all the edges of E exactly once. The Eulerian tour is constructed by starting from an arbitrary point and following edges until encountering the start point again. It is always possible to continue, because each time a node is visited, two incident edges are used, so it is an invariant of the process that all nodes have even degree. The process may encounter the start vertex before traversing all the edges, but in that case, just start again from a new vertex and piece the resulting Eulerian cycles together at the end, using the fact that the graph is connected.

FPT $O(2^k \cdot kn)$ for k-vc

if $k = \log_2 n$ $O(\log_2 n \cdot n^2)$

Best: $O(1.2738^k + kn)$