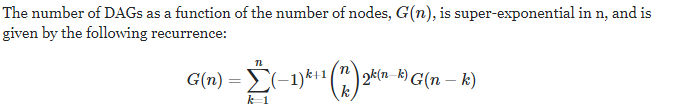
* We have a graph. If an edge has flow k then we have to pay k2 coins. Find the mincost maxflow.
* If an edge has cap c then replace it with c edges having weights 1, 3, 5, ..(2\*c -1 ). Find MCMF now.
* Given a gaph. Select a node or select all its adjacent edges. Find minimum (node + edges) to select s.t. every node is satisfied.
* The maximum-weight closure of a given graph G is the same as the complement of the minimum-weight closure on the transpose graph of G. Transpose is the graph having reverses edges.
* 
* **Matchings and Cycle Decompositions**

Given a weighted directed graph, decompose the graph into vertex disjoint cycles with maximum edge sum

Duplicate the nodes. Build a typical Bipartite graph from left to right. Use hungarian to find max cost max matching. There must need to be a Perfect matching.

* BPM but node I should be matched with eactly a\_i nodes. Easy. Use maxflow. Set weights wrt a\_i.
* For finding the edges which are in all maximum matching in BPM set matching edges directed forward and unmatched edges backward and find the edges which are not the same SCC
* For finding the nodes which are in all maximum matching in BPM

Sure, so we want to find all vertices v (let's say on the left side) for which there is a maximum matching in which v is not matched. To do this, find any maximum matching . Then run another phase where you look for alternating paths starting from unmatched vertices on the left. (This takes linear time). The answer is all vertices of the left side that were visited during this phase. (Clearly all unmatched vertices are part of the solution, and if there is some alternating path v 0 → w 0 → v 1 → w 1 → ... → v k from an unmatched vertex v 0 to a given vertex v k, then flipping the edges along that path gives us a maximum matching where v k is not matched

* BEST Theorem

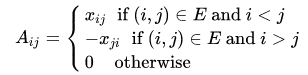
The BEST theorem states that the number ec(G) of Eulerian circuits in a connected Eulerian graph G is given by the formula



Here tw(G) is the number of arborescences, which are trees directed towards the root at a fixed vertex w in G. The number tw(G) can be computed as a determinant, by the version of the matrix tree theorem for directed graphs. It is a property of Eulerian graphs that tv(G) = tw(G) for every two vertices v and w in a connected Eulerian graph G.

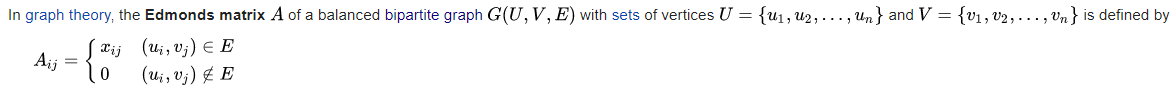
* Tutte Matrix

For a general graph If the set of vertices is V= {1, 2, 3, ..n} then the Tutte matrix is an n × n matrix A with entries



where the x[i][j] are variables. The determinant of this skew-symmetric matrix is then a polynomial (in the variables xij, i < j ) and is non-zero (as a polynomial) if and only if a perfect matching exists.

* Edmonds Matrix



where the xij are indeterminates. One application of the Edmonds matrix of a bipartite graph is that the graph admits a perfect matching if and only if the polynomial det(Aij) in the xij is not identically zero. Furthermore, the number of perfect matchings is equal to the number of monomials in the polynomial det(A), and is also equal to the permanent of A. In addition, rank of A is equal to the maximum matching size of G.

a monomial is a product of powers of variables with nonnegative integer exponents. X^2 + 2x^3y^2 + 1 has 3 monomials

* Find a permuation such that sum of f(p[i], p[i + 1]) is maximum where f(u, v) is the minimum weight on the path from u to v in a tree. Answer is just total sum of the weights. To find the permuation check my gomory hu tree implemented problem.
* Chinese Postman Problem Undirected

Chinese postman problem is to find a shortest closed path or circuit that visits every edge of an (connected) undirected graph. When the graph has an Eulerian circuit (a closed walk that covers every edge once), that circuit is an optimal solution. Otherwise, the optimization problem is to find the smallest number of graph edges to duplicate (or the subset of edges with the minimum possible total weight) so that the resulting multigraph does have an Eulerian circuit. Check my guthub for solution.

Directed:

On a directed graph, the same general ideas apply, but different techniques must be used. If the directed graph is Eulerian, one need only find an Euler cycle. If it is not, one must find paths from vertices with an in-degree greater than their out-degree to those with an out-degree greater than their indegree such that they would make in-degree of every vertex equal to its outdegree. This can be solved as an instance of the minimum-cost flow problem in which there is one unit of supply for every unit of excess in-degree, and one unit of demand for every unit of excess out-degree. As such it is solvable in O(|V|2|E|) time. A solution exists if and only if the given graph is strongly connected

* Find minimum number of edges on minimum weighted cut.

int main() {

ios\_base::sync\_with\_stdio(0);

cin.tie(0);

int n, m; cin >> n >> m;

Dinic F(n + 1);

for (int i = 1; i <= m; i++) {

int u, v, w; cin >> u >> v >> w;

F.add\_edge(u, v, (m + 1) \* w + 1);

}

long long a = F.max\_flow(1, n), k = a % (m + 1);

cout << (a - k) / (m + 1) << ' ' << k << '\n'; //(real flow, edges used)

return 0;

}

* Planar Graph

Euler's formula states that if a finite, connected, planar graph is drawn in the plane without any edge intersections, and v is the number of vertices, e is the number of edges and f is the number of faces (regions bounded by edges, including the outer, infinitely large region), then 

In a finite, connected, simple, planar graph for v >= 3

. If there are no cycles of length 3, then *e* ≤ 2*v* − 4.

*f* ≤ 2*v* − 4.

any planar graph must be 4 -colorable

Planar Dual Gaph:

Make every faces a vertex and for each edge add an edge between the faces that it divides with cost of that edge.

Mincut from s to t in a planar graph:

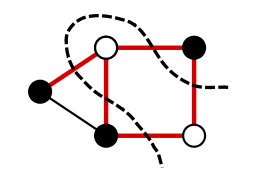
Add an additional face s..t.s and construct the dual graph. Shortest path from face(s) to face(t) is the answer. Check: <https://vjudge.net/problem/UVA-1376>

Maxcut in planar graph unweigted:

Note:

For a graph, a maximum cut is a cut whose size is at least the size of any other cut. The problem of finding a maximum cut in a graph is known as the Max-Cut Problem.

The problem can be stated simply as follows. One wants a subset S of the vertex set such that the number of edges between S and the complementary subset is as large as possible. Equivalently, one wants a bipartite subgraph of the graph with as many edges as possible.



Maxcut == m — minimum number of edges needed to be deleted to make it bipartite. Is is so because max cut of a bipartite graph is equal to the number of edges. Planar graph is bipartite iff all its faces are of even length. We take dual graph and we look at odd faces and we create a very similar graph to that one from chinese postman problem. We make a weighted clique on odd faces where weight of an edge is distance in dual graph. Such edge corresponds to deleting edges in original graph corresponding edges in dual graph connecting these two faces.

In short just create the dual graph, max cut = total edges – the edges that are doubled in an chinese postman problem of the dual graph of G.

Weighted solution is same.

* Given a DAG. Split it into k vertex disjoint paths such that sum of their length is maximum.

This problem can be solved with mincost k-flow algorithms. We build a network where each vertex of the graph is split into two (let's denote the vertices that we obtain when we are splitting some vertex i as v i, 1 and v i, 2). Then each directed edge transforms into a directed edge from vertex v i, 2 to vertex v j, 1 in the network, the capacity of this edge is 1, and the cost is 0. Also we add directed edges from the source to every vertex v i, 1 and from every vertex v i, 2 to the sink (they have the same characteristics: capacity is 1, cost is 0). And for each i we add a directed edge between v i, 1 and v i, 2; these edges actually represent that we are using some note in a melody, so their capacities are also equal to 1, and their costs are  - 1. The answer to the problem is equal to the absolute value of minimum cost of 4-flow in this network.

* Paging and Caching Problem:

There are some request for n types of books denoted as a[i]. You should satisfy them chronologically. Yout library can contain k types of books. You should buy the books everytime if it isn’t in the library. Cost of ith type is c[i]. Find minimum cost to satify all the requests.

We build a graph as follows. Let M be a huge number.

• We create two vertices 2i − 1 and 2i for each request i = 1, 2, ..., n, a source 0 and a sink 2n + 1.

• Every edge in the graph will have capacity 1.

• For each request i = 1, ..., n we create three edges:

– from the source to 2i − 1 with cost c[a[i]],

– from 2i − 1 to 2i with cost −M,

– from 2i to the sink with cost 0.

• For each two requests i < j we create an edge from 2i to 2j − 1 of cost 0 if a[i] = a[j] or of cost c[a[j]]

otherwise.

We look for a minimum-cost flow of value exactly k in this graph. We leave the correctness of this as an exercise; to begin, notice that any optimum solution must necessarily take all the edges (2i − 1, 2i) because their cost is hugely negative, and thus the solution is a minimum-cost collection of k paths from the source to the sink that together use all of those edges. Now think that we have k memory slots (or places in the library in which to store books). The i-th of these k paths corresponds to the sequence of requests that are served by elements stored in the i-th memory slot. The cost of this path corresponds to the cost of buyingthe books that occupy this slot.

Code:

int32\_t main() {

ios\_base::sync\_with\_stdio(0);

cin.tie(0);

int n, k; cin >> n >> k;

for (int i = 1; i <= n; i++) cin >> a[i];

for (int i = 1; i <= n; i++) cin >> c[i];

int s = 2 \* n + 1, t = s + 1, tmp = t + 1;

MCMF F(tmp); const int M = 1e9;

F.add\_edge(s, tmp, k, 0); F.add\_edge(tmp, t, k, 0);

for (int i = 1; i <= n; i++) {

F.add\_edge(tmp, i, 1, c[a[i]]);

F.add\_edge(i, i + n, 1, -M);

F.add\_edge(i + n, t, 1, 0);

for (int j = i + 1; j <= n; j++) {

if (a[i] != a[j]) F.add\_edge(i + n, j, 1, c[a[j]]);

else F.add\_edge(i + n, j, 1, 0);

}

}

long long ans = F.solve(s, t, k).second + 1LL \* M \* n;

cout << ans << '\n';

return 0;

}

* LR DAG is a DAG where each node of the original graph is represented by two nodes: a left node and a right node. There is an edge from a left node to a right node if there is such an edge in the original graph
* Minimum Vertex Disjoint Path Cover in a DAG: n – maximum matching in “LR DAG”.
* General Path Cover in a DAG: Now a node can be used multiple times. add some new edges to the LR graph so that there is an edge a → b when there is a path from a to b in the original graph (possibly through several edges).
* Minimum Edge Disjoint Path Cover in a DAG: 
* Minimum Vertex Cover in a Bipartite graph = Maximum matching
* Maximum Independent Set in a Bipartite Graph = n – maximum matching
* Maximum Weight Independent Set in a Bipartite graph:

This can be solved using maximum flow algorithm: construct a network where

source is connected with every left vertex by an edge with capacity equal to

weight of the vertex. then connect every left vertex to all right vertices that

are conflicting with this vertex by edges with infinite capacities; and then

connect every right vertex to the sink by an edge with its capacity equal to

value of the vertex (all edges have to be directed). Then the maximum power

of the deck is equal to sum - mincut, where sum is the sum of all weights and

mincut is the minimum cut value between the source and the sink (which is

equal to the maximum flow).

* Project and Instruments:

Petya has a simple graph (that is, a graph without loops or multiple edges) consisting of n vertices and m edges. The weight of the i-th vertex is ai. The weight of the i-th edge is wi. A subgraph of a graph is some set of the graph vertices and some set of the graph edges. The set of edges must meet the condition: both ends of each edge from the set must belong to the chosen set of vertices. The weight of a subgraph is the sum of the weights of its edges, minus the sum of the weights of its vertices. You need to find the maximum weight of subgraph of given graph. The given graph does not contain loops and multiple edges

cin>>n>>m; for(i=1;i<=n;i++) cin>>a[i]; ll s=0,t=n+m+1; ll sum=0; for(i=1;i<=m;i++){ cin>>u>>v>>w; flow.addedge(i,u+m,1e18); flow.addedge(i,v+m,1e18); flow.addedge(s,i,w); sum+=w; } for(i=1;i<=n;i++) flow.addedge(i+m,t,a[i]); cout<<sum-flow.max\_flow(s,t)<<nl;

* Minimum Edge Cover in a genaral Graph: n – maximum matching

A smallest edge cover can be found in polynomial time by finding a maximum matching and extending it greedily so that all vertices are covered.

* Minimum Weight Edge Coverin a genaral grah:

Given an undirected weighted graph G, find a minimum weight subset E such that each vertex is incident on at least one edges in E.

-> First, take all negative edges. Let's delete all vertices covered by them. If there was an edge connecting uncovered and covered vertex, let's say that now it connects uncovered vertex to itself. Now all edges are nonnegative.

ans+=sum of all negative edges

Let's look at the answer. If there is a path of length at least 3 then we can

delete the middle edge on this path and our answer will improve. It means in

optimat answer each connected component is a star(possibly with a single

edge). For each star let's choose one edge in it. Then chosen edges form a

matching. For all vertices not in this matching it's optimal to choose the

shortest edge incident to that vertex. Let's build 2 copies of the graph (not LR graph, we should literally duplicate them with corresponding edges being duplicated) and for

each vertex v, connect v0 and v1 with an extra edge with weight equal to the

shortest edge incident to v multiplied by 2. Then the answer is min weight

matching divided by 2.

ans+=min weight matching.

* Any cycle Cover in an undirected bipartite graph:

Create flow graph but source and sink edges will have flow 2. Check if (left part nodes == right part nodes && max flow ==2 \* left part nodes)

* Min cost cycle cover in a general undirected graph:

It assumes a cycle cover exists.

A cycle cover of an undirected graph is a collection of vertex-disjoint cycles such that each vertex belongs to exactly one cycle. We require each cycle contains at least three vertices. In a weighted graph, the weight of a cycle cover is the sum of the weights of all its edges. Find the minimum total weight cycle cover.

For each edge v - u let's create separate copies of vertices v and u. For each vertex v our matching should contain exactly 2 edges incident to v. Let's create deg(v) - 2 additional vertices and connect them to all copies of v we created with 0 weight edges. Now perfect matching will match those deg(v) - 2 vertices to deg(v) - 2 copies of v and other 2 copies will correspond to edges in cycle cover. If perfect matching doesn’t exist then no cycle cover exists.

* Shortest Path with even or odd number of edges:

Given an undirected positive weighted graph G, find the shortest simple path

from s to t with even number of edges.(OR odd edges)

->

Let's color edges on a path in black and white alternatingly. Then black edges form a matching covering all vertices on a path except s and white edges form a matching covering all vertices except t. Let's build 2 copies of our graph, G 0 and G 1 for black and white matching respectively. For each v ≠ s, t v is either in both matchings or in neither. To achieve it, let's add edge v 0 - v 1 with zero cost. s should not be in black matching, so let's add new vertex s 2 and edge s 0 - s 2. s should be in white matching, so no new edges for s 1 needed. For t by similar logic we should add vertex t 2 and edge t 1 - t 2. Min cost perfect matching in this graph corresponds to shortest path with even number of edges. By the way, if we add edge t 0 - t 2 instead t 1 - t 2 we will find shortest path with odd number of edges

* Given a DAG. You can remove at most 2 vertices with zero in degree in each round. How many rounds do you need to remove all the vertices?

Put an edge iff two vertices are incomparable(u is not reachable from v and vice versa) and answer is n — max matching.

* Two matching. Given an undirected weighted graph G, find a minimum weight subset E such that the degree of each vertex in the subset E is no more than 2.

For bipartie graph just use mcmf with sink and sources edges having flow = 2.

Otherwise exactly min cost cycle cover solution will do the work.

* Hall’s Theorem

Hall’s theorem can be used to find out whether a bipartite graph has a

matching that contains all left or right nodes. If the number of left and right

nodes is the same, Hall’s theorem tells us if it is possible to construct a perfect

matching that contains all nodes of the graph. Assume that we want to find a

matching that contains all left nodes. Let X be any set of left nodes and let f

(X) be the set of their neighbors. According to Hall’s theorem, a matching that

contains all left nodes exists exactly when for each X, the condition |X| ≤ |f

(X)| holds.

where |X| is the size of the set X.

* POSET

An antichain is a set of nodes of a graph such that there is no path from any node to another node using the edges of the graph.

Maximum Antichain:

Dilworth’s theorem states that in a directed acyclic graph, the size of a minimum general path

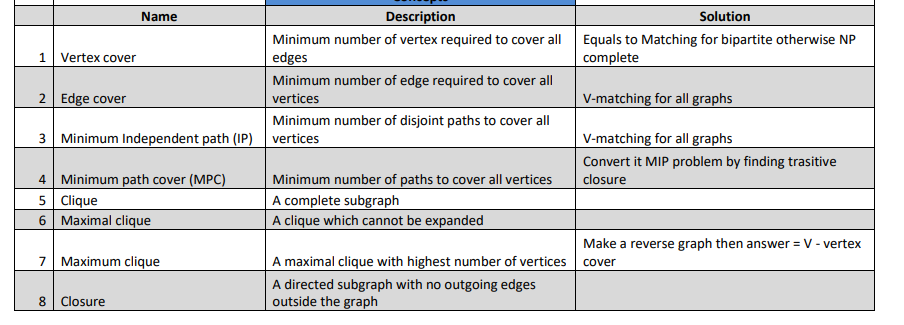
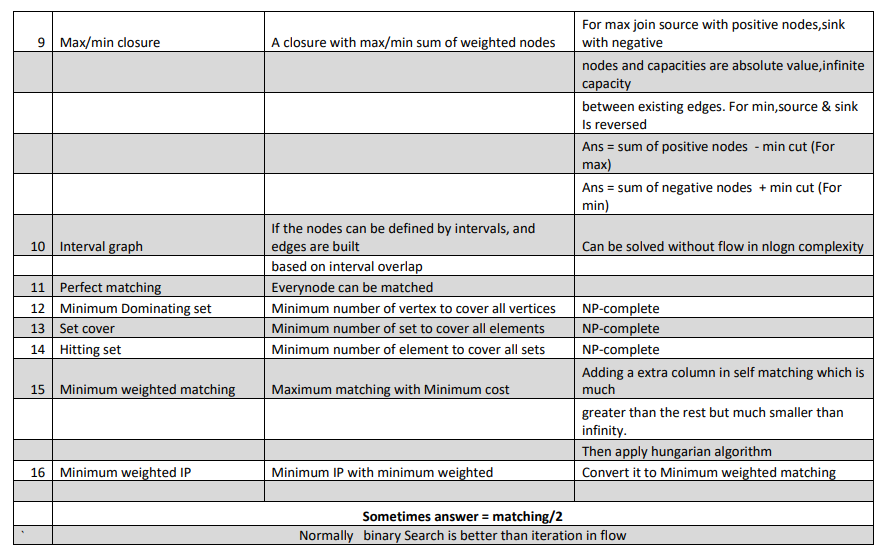
cover equals the size of a maximum antichain.

Mirsky’s Theorem: Minimum number of antichains the poset can be partitioned = maximum length of a chain. Construction: let dp[x] = maximum length of a chain which ended in this node. Then the nodes in bucket i will be the nodes such that dp[x] = i

Minimum number of chains the poset can be partitioned = maximum antichain

Note that Mirsky's theorem can be restated in terms of directed acyclic graphs (representing a partially ordered set by reachability of their vertices

Erdős–Szekeres theorem: in every partially ordered set of rs + 1 elements, there must exist either a chain of r + 1 elements or an antichain of s + 1 elements

* Mengers Theorem: the size of a minimum cut set is equal to the maximum number of edge disjoint paths that can be found between any pair of vertices. For vertex disjoint paths just split the vertex
* 
* 
* Determining the uniqueness of a min-cut:

The cut is unique iff there is no other min-cut.

If you succeed in finding a different min-cut, then the first min-cut isn't unique.

Your link gave us one min-cut, which is all the reachable vertices from s in the residual graph. Can you think of a way to obtain a different cut, not necessarily the same?

Why did we take those vertices reachable from s in particular?

Maybe we can do something analogous from t?

Look at the same residual graph, starting at t. Look at the group of vertices reachable from t in the reverse direction of the arrows (meaning all the vertices which can reach t).

This group is also a min-cut (or actually S \ that group, to be precise).

If that cut is identical to your original cut, then there is only one. Otherwise, you just found 2 cuts, so the original one can't possibly be unique.