Graph Theory

Jingbang Chen

Zhejiang U Legilimens

Outline

Basics

- BFS
  - *How to apply two-way BFS?*
  - *How to apply 0-1 BFS?*
- Dijkstra
- Bellman-Ford, SPFA
- Floyd-Warshall
  - *How to explain it in a dynamic programming way?*

Difference Constraints System

- Bellman-Ford, SPFA
- How to add edges?
- How to add $'<'$ or $'>'$?
- What does a negative cycle mean?
- Don't forget the hidden restriction

Dijkstra

- Can find the shortest path from multiple starting points
- Can solve K-shortest Path
- Can find optimal solution when the cost is not-decreasing and have non-aftereffect property
- How to understand each iteration/expansion and its order?

Shortest Path Graph/Tree

- Direct Acyclic Graph
- Tree: Any Traversal Tree of it
- Classic Example: Maintain the shortest path if an adjoining edge is prohibited

K-Shortest Path

- Second Shortest: Modified Dijkstra(mentioned before)
- General Approach: A* Search

A* Search Steps

1. Calculate the shortest path between any i to T as $h[i]$.
2. $f[i] = d[i] + h[i]$, $d[i]$ means the distance from S for this i.
3. Add S into a priority queue Q.
4. Pop out the x with the smallest $f[x]$.
5. If $x = T$ and it is the K_{th} time, return $d[x]$ as the answer.
6. Traverse all adjoint points and add them into Q.
7. Goto 4.

Basics

- Strong Connected Components
- 2-Edge-Connected Components

  without any bridge.
- 2-Vertex-Connected Components

  without any articulation vertices.
- Tarjan Algorithm for everything?

Tarjan

- Strong Connected Components: Just apply it.
- 2-Edge-Connected Components

  Don't visit the direct father.

- 2-Vertex-Connected Components

  Don't visit the direct father. While enter a new edge $t(x, y)$, add t to the stack S. After visiting y, if $low[y] >= dfn[x]$, a new BCC can be found by popping edges from S until an edge (x, y) has been popped.

Tarjan

- Another way: Just remove all bridges or articulation points and do flood-fill.
- Bridges: for an edge (x, y), if $low[y] >= dfn[x]$, it is a bridge
- Articulation points: for an edge (x, y), if $low[y] >= dfn[x]$ and x is not the root, y is an articulation point. If x is the root and it has more than 1 child, it is also an articulation point.
- Another another way: Just find all components and trace back.

Dominator Tree

- A tree with S as the root.
- For any node i, every node on the path (S, i) must be passed if we want to go to i from S.
- For any node i, S must pass i if it wants to go to any node t belong to i's subtree.
- $O(n\alpha(n))$

Dominator Tree

- Property for a traversal tree: If $dfn[v] \leqslant dfn[w]$, any path from v to w must contain a common ancestor in the tree.
- Semi-Dominating node(semi[x]):$semi[x] = min(v|P = (v = v_0, v_1, ..., v_k = x), dfn[v] > dfn[x])$
- Dominating node(idom[x]): the deepest dominating node.
- $deep[idom[x]] \leqslant deep[semi[x]]$(on the path (S, x))
- If v is w's ancestor, v is $idom[w]$'s or $idom[w]$ is $idom[v]$'s.

Lengauer-Tarjan Algorithm

1. Construct the traversal tree.
2. Calculate $semi[x]$ for every node x.
3. Calculate $idom[x]$ by $semi[x]$.

Query for Connectivity While Deleting Edges

- 1.Construct a traversal tree.
- 2.Give every non-tree edge a random value.
- 3.Every tree edge's value equals to the Exclusive OR of all non-tree edges which cross over it.
- 4.For each query, if there exists a subset of edges which the Exclusive OR Result is 0, the graph is not connected.

  The result proves that a tree edge and all related non-tree edges are selected. We can calculate the linear basis to check.

- We can delegate 2^i to non-tree edges if possible instead of random value.

Dynamic Maintain the Number of Bridges

- Operations: Add or Delete an edge each time.
- Offline all operations and apply a CDQ Divide and Conquer.
- O(nlogn)

Dynamic Maintain the Number of Bridges

Dynamic Maintain the Number of Bridges

0.$Solve(l, r, n, m)$ means considering all operations in $[l, r]$, in a graph with n nodes and m edges.
1.If $l = r$, apply Tarjan directly.
2.Select all edges E modified in $[l, r]$, $|V| = O(|E|)$.
3.Add all edges other than E, since they will remain unchanged.
4.Contract all 2-Edge Connected Components and build a forest. All non-tree edges can be deleted.
5.Build virtual trees based on non-added edges. Every edge on virtual trees represents a series of bridges.
6.$O(|V'|) = O(|E'|) = O(r - l)$. $Solve(l, mid, n', m')$. Apply all operations in $[l, mid]$, then $Solve(mid + 1, r, n', m')$.

Dynamic Maintain the Number of BCCs

- Operations: Only add edges.
- General: Contract all BCCs to a node and use Link-Cut Tree to maintain the compressed tree.
- If x and y are not connected, adding (x, y) will only link them. Just do it on LCT.
- If they are connected but belong to different components, enumerate and contract all nodes on path $(belong[x], belong[y])$ on LCT.
- Use Disjoint Set to maintain the relationship and connectivity.

Basics

- Prim
- Kruskal
- Link-Cut Tree

Matrix-Tree Theorem

- Undirected Graph: The answer should be the determinant of the matrix gotten by removing any i_{th} row and column.
- $$A_{i,j} = \begin{cases} -C & \text{C-connected, } i \neq j \\ 0 & \text{unconnected, } i \neq j \\ \sum_{k=1}^{n} -A_{i,k} & i = j \end{cases} \quad (1)$$
- if C means the weight of edge, the answer will become the sum of all possible spanning tree's $\prod C$.

Matrix-Tree Theorem

- Directed Graph: The number of oriented spanning trees(i can reach all nodes) rooted at a node i is the determinant of the matrix gotten by removing the i_{th} row and column.
- $$A_{i,j} = \begin{cases} -C & \text{C-connected, } i \neq j \\ 0 & \text{unconnected, } i \neq j \\ \sum_{k=1}^{n} -A_{k,i} & i = j \end{cases} \tag{2}$$
- if C means the weight of edge, the answer will become the sum of all possible oriented spanning tree's $\prod C$.
- change $-A_{k,i}$ to $-A_{i,k}$ if it is reverse-oriented(All nodes can reach i).

Steiner Tree

- f[mask][x]
- f[][] initialize to be *inf* while f[1<<i][Ai]=0
- $f[mask][x] = min(f[mask][x], f[sub][x]+f[mask\ xor\ sub][x])$
- $f[mask][x] = min(f[mask][x], f[mask][y] + minD(y, x))$

SPFA may get TLE, so we can use Dijkstra instead.

MST on Planar Graphs

- Manhattan Distance: For each point x, divide the plane into 8 regions by 45 degree. Only edges between x and the closest point in each region are needed.
- Euclidean Distance: Compute the Delaunay triangulation and only edges in the triangulation are needed.
- Easier way to understand: Divide the plane into 6 regions by 60 degree instead.(But you cannot do it without the Voronoi diagram, which is the dual planar graph for the Delaunay triangulation)

MST on Random Planar Graphs(Euclidean)

- Method 1: Do it as if no random at all.
- Method 2: Choose some random direction and sort points by projection of point to it. For each point, enumerate and maintain p closest points until the current point's projection distance is longer than all chosen points's actual distance.
- Method 3: Divide the plane into several square regions, enumerate regions around it in a suitable distance for each point.(Usually in sqrt(MAXX))
- Method 4: Use KD-Tree to find the closest few points for each one. Then apply Kruskal.
- Method 5: Use KD-Tree to maintain the closest points for each one. Then apply Prim.

Degree-Constrained Spanning Tree

- Minimum Degree Spanning Tree: NP hard.
- Determine maximum *degree* $\leqslant k$: NP-completeness.
- Only restrict the root node *degree* $\leqslant k$: A classic problem.

Algorithm for the above problem

1.Remove all edges of the root and find the spanning forest.
2.If there are p components, $p <= k$ must hold. Then connect each component to the root, which build a p-degree solution.
3.Try to add an used edge (root,y) and delete the largest edge on the path (root,y) which can maintain by a DFS each time. Find the minimum way and build a (p+1)-degree solution.
4.Repeat 3 to get a k-degree solution.

Minimum Spanning Arborescence

- Chu-Liu/Edmonds' Algorithm: O(nm).
- Can be improved to O(m+nlogn).
- No determined root: Construct a fake one.

Chu-Liu/Edmonds' algorithm

1.Find the smallest in-edge for each node except the root.
2.If no cycle exists, the answer has been found. Otherwise, for each cycle C, contract it into one node c.
3. Delete all edge $(u, v \in C, w)$; change $(u \in C, v \notin C)$ into (c, v, w); change $(u \notin C, v \in C, w)$ into $(u, c, w - w(in(x)))$. $in(x)$ denotes the in-edge of x.
4. Find the minimum Spanning Arborescence. For each used edge (u,c), we can find the deleted edge to break C.

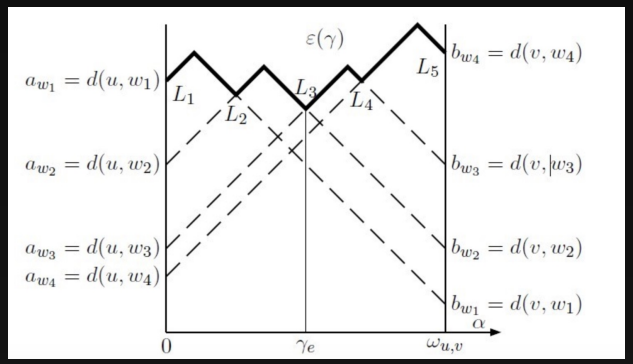
Minimum Diameter Spanning Tree

- Find the Absolute Center C of G: minimize $max(dist(C, i)), i \in G$.
- Find the Shortest Path Tree from C.

Minimum Diameter Spanning Tree

Find the Absolute Center of G

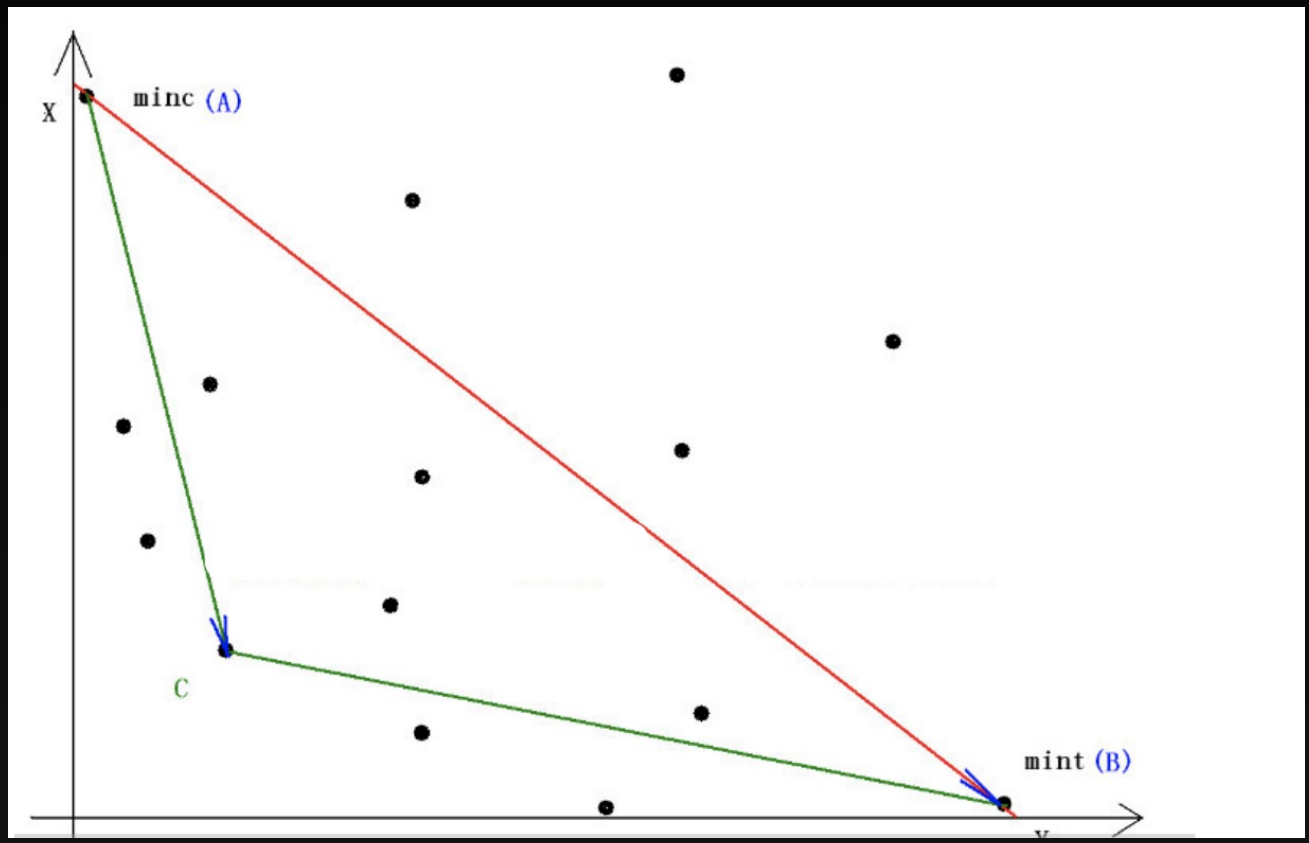
1.Suppose the Center is on edge (u, v, w).
2.for node i, define $f(i) = min(dist(u, i) + x, dist(v, i) + w - x$. Choose the best x to minimize $max(f(i))$. Also see the below image, the best x must be some broken line's turn point, which means $dist(u, i) = dist(v, i) + w = y$, so just sort all $dist(u, i)$.
3.Enumerate all edges to find the best one.



Minimum Product Spanning Tree

- Each edge has two value a, b, find a spanning tree that minimize $\sum a * \sum b$.
- Denote $\sum a$ as x, $\sum b$ as y, We should find minimum $k = xy$, which means $y = \frac{k}{x}$ should be as close to the axis as possible. It must on a lower convex hull.
- Find minimum x and y points as A and B(Apply Kruskal for only a or b).
- Divide and Conquer, find the furthest C and divide it into two subtasks, which means maximize $S_{A,B,C}$.
- Change value to $b * (B.x - A.x) + (A.y - B.y) * a$ and apply Kruskal to find C.
- Work(A,C) and Work(C,B). Divide until no other possible point(the cross product of BA and CA is non-negative).

Minimum Product Spanning Tree



K Smallest Spanning Tree

- General idea: If we know about all $(i-1)_{th}$ smallest spanning tree, we may be able to find i_{th} smallest spanning tree by simply replacing one of edges.
- Time Complexity: $O(KM\alpha(M,N)+MlogM)$
- Reference: Harold N. Gabow, Two Algorithms for Generating Weighted Spanning Trees in Order, SIAM J.COMPUT. Vol. 6,No. 1, March 1977
- Other Algorithms exists, which is usually a modification of it. Most algorithms have better time complexity of planar graphs.

K Smallest Spanning Tree

$EX(T, IN, OUT)$

For a spanning tree T, $EX(T, IN, OUT)$ finds the smallest T-exchange e, f of weight r when $e \in T - IN, f \in G - OUT$.

$GENK(K)$

1.Use a global priority queue Q to maintain tuple (t, e, f, F, IN, OUT) as P_j^i, where t is the weight of the smallest tree under certain restriction generated by a T_i-exchange e, f, F is the father of T_i, IN and OUT are lists of edges.
2.Find a minimum spanning tree T_1 and its father array F. After $EX(F, \phi, \phi)$, add $(t + r, e, f, F, \phi, \phi)$ into Q.
3.Applying GEN for $K - 1$ times to find $T_2..T_K$.

K Smallest Spanning Tree

GEN

1.Remove the tuple (t, e, f, F, IN, OUT) with smallest weight t from Q.
2.modify F to F_j so edge f replaces e, which is the father array of the next spanning tree.
3.$t_i = t - w(f) + w(e)$, modify IN to IN_i by adding e to IN, modify OUT to OUT_j by adding e to OUT.
4.$EX(F, IN_i, OUT)$ and add $(t_i + r, e, f, F, IN_i, OUT)$ to Q.
5.$EX(F_j, IN, OUT_j)$ and add $(t + r, e, f, F_j, IN, OUT_j)$ to Q.

Basics

- A subset of vertices of an undirected graph such that every two distinct vertices in the clique are adjacent .
- Maximal Clique

  A clique that cannot be extended by including one more adjacent vertex.

- Maximum Clique

  The largest clique in the graph.

- Opposite: Independent Set.
- Finding a maximum clique or all cliques are NP-Complete.

Maximum Clique Finding

- Searching with pruning.

A Classical Method

$S_i = \{V_i, V_{i+1}, ..., V_n\}$, $mc_i = MC(S_i)$.
Obviously $MC(V) = mc_1$, $mc_i = mc_{i+1}$ or $mc_i = mc_{i+1} + 1$.
The latter situation only occurs when there exists a clique containing v_i.
Several Pruning way:
1.current-size+remain-vertex $\leqslant$ ans
2.current-size+mc[i] $\leqslant$ ans

Maximal Cliques Counting

- Bron-Kerbosch Algorithm

```
BronKerbosch1(R, P, X):
    if P and X are both empty:
        report R as a maximal clique
    for each vertex v in P:
        BronKerbosch1(R ∪ {v}, P ∩ N(v), X ∩ N(v))
        P := P \ {v}
        X := X ∪ {v}
```

Maximal Cliques Counting

- Shortage: Inefficient when the graph has many non-maximal cliques.
- Modification 1: Choose a pivot u.
- Maximal Clique should contain either u or one of its non-neighbors.
- $u \in P \backslash N(u)$

```
BronKerbosch2(R,P,X):
    if P and X are both empty:
        report R as a maximal clique
    choose a pivot vertex u in P ∪ X
    for each vertex v in P \ N(u):
        BronKerbosch2(R ∪ {v}, P ∩ N(v), X ∩ N(v))
        P := P \ {v}
        X := X ∪ {v}
```

Maximal Cliques Counting

- Modification 2: Vertex Ordering
- Degeneracy: The smallest number d such that every subgraph has a vertex with degree d or less.
- Degeneracy ordering: Repeatedly selecting the vertex of minimum degree among the remaining vertices.
- the set P of candidate vertices in each call will be guaranteed to have size at most d.

```
BronKerbosch3(G):
    P = V(G)
    R = X = empty
    for each vertex v in a degeneracy ordering of G:
        BronKerbosch2({v}, P ∩ N(v), X ∩ N(v))
        P := P \ {v}
        X := X ∪ {v}
```

Maximal Cliques Counting

- Worst-case Analysis: $O(3^{n/3})$

  A graph has at most $3^{n/3}$ maximal cliques.

- With all modification: $O(dn3^{d/3})$.

  d-degenerate graphs has at most about $(n-d)3^{d/3}$ maximal cliques.

Basics

- Euler: Pass every edge exactly once.
- Hamilton: Pass every node exactly once.
- Path and Circuits.

Euler Path and Circuits

- Change Path to Circuit: Connect two special nodes first.
- Fleury's Algorithm: Keep going until all edges are passed. When there is a non-edge choice, must pass it first.

  What is its truly time complexity?

- Hierholzer's Algorithm: Keep extending new cycle.

  How to implement? Current-Edge Optimization!

```
void dfs(int x){
    for(int&i=g[x];i;){
        if(vis[i]){i=nxt[i];continue;}
        vis[i]=vis[i^1]=1;
        int j=w[i];
        dfs(v[i]);
        ans[++cnt]=j;
    }
}
```

Euler Path and Circuits

- Mixed Graph: Decide the direction for all undirected edges, then do as normal.

  for any x, $d_x = (|out\_degree - in\_degree|)/2$.

  Add edge (S, x, d_x) if $out\_degree > in\_degree$. Add (x, T, d_x) otherwise.

  Add all undirected edges with $flow\_limit = 1$.

  Apply MaxFlow and check if every node is full of flow.

Hamilton Path and Circuits

- Dirac Theorem: If every node has a degree more than $floor(n/2)$, it must have a Hamilton Circuit.

  How to construct a circuit under this condition?

- Tournament Graph must have a Hamilton Path while strong connected one has a circuit.

  How to construct a path under this condition?

Best Theorem

- A directed graph $G = (V, E)$ and every node v has degree $deg(v)$.
- $$ec(G) = t_w(G) \prod_{v \in V} (deg(v) - 1)!$$
- $t_w(G)$ indicates when w is root, the number of oriented spanning tree. For a Euler Graph, $t_w(G) = t_v(G)(w, v \in V)$. We can calculate it by using Matrix-Tree Theorem.
- Notice: the result doesn't fix the start point and multiple edges will be considered different edges.

Route Inspection Problem

- AKA Chinese postman problem: Find a shortest closed path or circuit that visits every edge of a graph .
- Basic Idea: Add extra edges representing paths and find Euler Circuit.
- Undirected Graph: Find all nodes with odd degree(Must have even numbers). Calculate the shortest path between any two of them. Find the minimum weight matching and add matching edges.
- Directed Graph: Find all nodes $in\_degree \neq out\_degree$. Then apply the similar method.

Basics

- Dinic/SAP
- Edmonds-karp with Shortest Path Algorithms
- Max-Flow Min-Cut Theorem

Min-Cut

- What's the meaning of cut?
- Just well-define the meaning of Set S and T.
- Pay attention to the dependency.
- Use *inf* to prevent some edges from affecting the decision.
- Use edges related to S or T to deal with nodes' value.
- How to Print solutions?

  Cannot just select full-flow edges. DFS in the residual graph to find S and T. Those edges connect S and T are the answer.

- Applications of Minimum Cut Model in Informatics
  Amber Hu, 2007

Maximum Density Subgraph

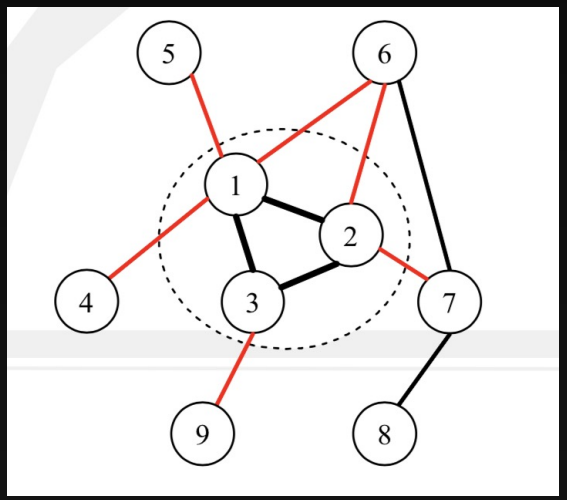
- $Density = \frac{Number\ of\ edges}{Number\ of\ nodes}$
- Typical Maximum Weight Closed Subgraph

  What does 'Closed' mean?

- Solution: Binary search and solve it as a MWCS problem.
- Create nodes for edges and nodes and add dependency that (x,y) $->$ x,y.

Maximum Density Subgraph

- Target: Maximize $|E'| - g * |V'|$
- Property: For any $|V'|$, $|E'|$, choose edges of Induced graph of V' will maximize the target function.
- See the below image, Maybe we can select red edges($x \in V', y \notin V'$) first and get E' by V' total degree.



Maximum Density Subgraph

- These edges are cut between V' and $\overline{V'}$.

$$\text{Maximize} \quad |E'| - g \cdot |V'| = \sum_{e \in E'} 1 - \sum_{v \in V'} g$$

$$\Updownarrow$$

$$\text{Minimize} \quad g \cdot |V'| - |E'| = \sum_{v \in V'} g - \sum_{e \in E'} 1$$

$$= \sum_{v \in V'} g - \left(\frac{\sum_{v \in V'} d_v}{2} - c[V', \overline{V'}] \right)$$

$$= \sum_{v \in V'} (g - \frac{d_v}{2}) + c[V', \overline{V'}]$$

Maximum Density Subgraph

- Target: Minimize $\sum_{v \in V'} 2g - d_v + 2c[V', \overline{V'}]$
- So choose v will cost $2g - d_v$.

Network Construction

1.change undirected edge (u, v) to two directed edges $\langle u, v \rangle$ and $\langle v, u \rangle$.
2.Connect S with all $\langle u, v \rangle$ of U.
3.Connect all v to T of $U + 2g - d_v$.
4.U for making flow limit become non-nagative.

- Generalization to Both Node and Edge Weighted Graph: Similar derivation reaches $c(v, t) = U + 2(g - p_v) - d_v$

Cost Flow

- It is so fucking hard though.
- Many thanks to Kunwei Zhang, Tsinghua University. Everyone should read his blog as carefully as possible.

Cost Flow

- Minimum Cost Maximum Flow

  Just keep finding argument path.

- Minimum Cost Flow

  Stop when the argument path is of negative profit.

Cost Flow

- Things will fuck our cost flow:

  Negative edges, cycles, surplus and deficit

- Surplus and deficit: Minimum Cost Feasible Flow

  Create S' and T', for node i, add $(S', i, 0, e_i)$ if the surplus value $e_i > 0$. Otherwise add $(i, T', 0, -e_i$. Then Apply MCMW. If the flow equals $\sum e_i$, the feasible flow exists and it shares the same residual network.

Cost Flow

- Things will fuck our cost flow:

  Negative edges, cycles, surplus and deficit

- How to delete the original S and T:

  Add an edge $(T, S, 0, inf)$.

- Surplus and deficit: Find feasible flow first

  Create S' *and* T'*, for node* i*, add* $(S', i, 0, e_i)$ *if the surplus value* $e_i > 0$*. Otherwise add* $(i, T', 0, -e_i)$*. Then Apply MCMW. If the flow equals* $\sum e_i$*, the feasible flow exists and it shares the same residual network. Then we can apply necessary algorithms from* S *to* T *if necessary.*

Cost Flow

- Only negative edges but not cycles:

  Sometimes we can just ignore them. Or we can change the cost to $c_{i,j} - D_i + D_j$, D_i means the shortest distance to T, which can be easily traced back and won't change the optimal property.

- Negative cycles exists:

  Apply Cycle Elimination Algorithm(Usually not necessary). Or we force all negative edges full of flow. In this way, we break the balance condition but hold the optimal condition. Then we find feasible then maximal flow steps by steps.

Cost Flow

- ZKW Cost Flow: Faster in Constraints in some network. Not allowed with negative edges(Well, we have discussed many methods to eliminate them)

  More Specifically, it will do well in networks with short argument paths or larger flow or limited range in cost value.

- *Prim-Dual: Combine two algorithms' advantages.
- Won't discuss much, please refer to Kunwei Zhang's blog or others for further information.

Flow with Lower-Bounds

- Cost Flow: Discuss it before.
- Feasible Flow: Similar to what have been discussed before.

  Add (T, S, inf) to change it to cycle flow model. Then add (S', i, e_i) if the surplus value $e_i > 0$. Otherwise add $(i, T', -e_i)$. Then Apply MaxFlow Algorithm. If the flow equals $\sum e_i$, the feasible flow exists and it shares the same residual network.

- Maximum Flow: Find feasible flow first then apply MaxFlow algorithm from S to T.
- A simple Trick: Separate the edge (x, y, low, max) to (x, y, low) and $(x, y, max - low)$. Give the first one -inf cost and apply Cost Flow.

Stoer-Wagner Algorithm

- Target: To solve the minimum cut problem in undirected weighted graphs with non-negative weights.
- Find any S-T minimum cut in the graph and contract S and T then. Keep repeating it until only one node in the graph.
- Property: For any S and T, they are either in the same side or the different side of the minimum cut. So the result can only be minimum cut between S and T or the cut in the new graph.

Stoer-Wagner Algorithm

MinimumCutPhase()

1.Choose an arbitary node a and Set $A = a$ initially.
2.Keep adding nodes to A until $A = G$. Every step we choose x that has the maximum $w_{x,y}, x \notin A, y \in A$.
3.Define the last two nodes being added as S and T. The minimum cut is $w_{x,y}$ when T being added.
4.Replace S and T with a new node u, $w_{u,x} = w_{s,x} + w_{t,x}$.

MinimumCut()

When $|G| > 1$, Apply MinimumCutPhase().
Use the new cut result to update the answer.

Gomory-Hu Tree

- Construct a tree that the minimum cut between any two nodes x, y in the original graph equals to the minimum edge of path (x, y).
- Property: Any two nodes x, y have at most $n - 1$ kinds of minimum cut on the graph.

Gomory-Hu Tree

1. Choose any x, y and find their minimum cut $[S, T]$ of cost w.
2. Add an edge (x, y, w).
3. Solve S and T separately.

Minimum Cut Uniqueness

- Minimum Cut Uniqueness:

  Every node i can be either connected by S or by T on the residual network.

- Can Edge (x, y) occur in some minimum cut?

  Calculate SCC on the residual network, iff id[x]=id[v]. Because the new graph only contains full-flow edges, so that any $S-T$ Cut on the new graph can correspond to any minimum cut in the original graph.

- Must Edge (x, y) occur in some minimum cut?

  iff id[S]=id[u] and id[v]=id[T]. If increase the limit of (x, y) , there will be an argument path S–x–y–T.

Basics

- Maximum Matching on Bipartite Graph: Hungary, Hopcroft
- Maximum Weight Matching on Bipartite Graph: Kuhn-Munkres Algorithm
- Maximum Matching and Maximum Weight Matching: Blossom Algorithm
- http://www.csie.ntnu.edu.tw/~u91029/Matching.html

Basics

- Maximum Matching on Bipartite Graph: Hungary, Hopcroft
- Maximum Weight Matching on Bipartite Graph: Kuhn-Munkres Algorithm
- Maximum Matching and Maximum Weight Matching: Blossom Algorithm
- http://www.csie.ntnu.edu.tw/~u91029/Matching.html

Classical Problems Revisited

- In Bipartite Graph: Minimum Path Cover = Maximum Independent Set = N - Maximum Matching
- In Bipartite Graph: Minimum Vertices Cover = Maximum Matching
- In any graph: Maximum Independent Set + Minimum Vertices Cover = N
- In DAG: Minimum Path Cover = N - Maximum Matching

  For an edge (x, y), add (x, y) to the graph for matching.

Classical Problems Revisited

- In Bipartite Graph: Minimum Path Cover = Maximum Independent Set = N - Maximum Matching
- In Bipartite Graph: Minimum Vertices Cover = Maximum Matching
- In any graph: Maximum Independent Set + Minimum Vertices Cover = N
- In DAG: Minimum Path Cover = N - Maximum Matching

  For an edge (x, y), add (x, y) to the graph for matching.

Hall's Marriage Theorem

- G is a bipartite graph (A, B).
- If for any $X \subseteq A$, $|N(x)| \geqslant |X|$, there must be a matching cover A.
- This theorem can hold multiple edges.

Hall's Marriage Theorem

- K-regular: There exists k perfect matching with edge-indepedently. Just find perfect matching for k times and delete edges. It can also hold multiple edges.
- 2-Factor Theorem: For any 2k-regular graph, it can be decomposed to k edge-independent subgraph which contains all nodes and their degree $= 2$.

  Find a Euler Circuit, then divide each node i to i and i'. Each edge on the circuit (i, j) becomes (i, j'). The new graph becomes a bipartite graph with degree $= k$. Then find the matching.

Basics

- Multiple ways calculating LCA.
- Heavy-Light Decomposition
- Divide and conquer on tree.
- Mo's Algorithm on tree.
- Virtual tree

DSU on tree

- Only contains queries for subtrees and without any modifications.
- Take advantage of Heavy-Light Decomposition's properties.
- When calculate node x, add every necessary information by Brute-Force. Only eliminate the effect when it is not a heavy-son. Then add light-sons' information again.
- Only merge subtree to upper-level heavy chain when we pass on the light-edge. There are no more than *logn* heavy chains in a tree, so that every node will only be merge at most *logn* times.

Prufer Order

- A sequence correspond to a unrooted tree.
- the Length will be $n-2$ for a tree with n nodes.
- Process: Find the smallest node with $degree = 1$. Delete it and add the node conncected to it into the sequence. Repeat this operation.
- Reverse: Select the from element, Find the smallest v which hasn't occur in the sequence. Add edge (u, v) and delete them. At last,Connect the remain two nodes.

Prufer Order

- Each node will occur $degree - 1$ times.
- Cayley Formula: A complete graph has n^{n-2} way to generate trees.
- Extended Cayley Formula: A complete graph has $F(n, m) = m * n^{n-1-m}$ ways to generate forests of m trees.

Long-Chain Decomposition

- Find the long-son instead of heavy-son.
- Each time, the node will inherit the information only from the long-son. Other sons should be traversed to add information.
- Similarly, Each node will only be inherited once and be recalculated when it comes to a heavy edge.
- Easy to understand but quite hard to code, with complexity of $O(n)$.

Basics

- 2-SAT
- Big-Small
- Meed in the Middle

Cycles Enumeration

Cycle Length = 3, $O(m\sqrt{m})$

1.Choose the direction for each edge: If $deg[x] > deg[y]$ or $deg[x] = deg[y]$ and $x < y$, x will point to y.
2.Enumerate each node x, then enumerate all its out-edges and mark the other node as y.
3.Enumerate all out-edges of y, if the other node as z is marked, (x, y, z) will be a cycle.

- When cycle length $= 4$, it also requires calculating of cycle length $= 3$. Then after the further enumeration, we should also use the Inclusion-Exclusion Principle.

Chordal Graph

- All cycles of four or more vertices have a chord that connects two node of the cycle.
- Perfect Elimination Ordering: For each vertex v, v and the neighbors of v that occur after v in the order form a clique. A graph is chordal iff it has a perfect elimination ordering

  Start from backwards. Each time choose the node with largest value. Then increase all non-selected neighbor nodes by 1. Need to validate the property.

- Maximal Cliques Number = Minimal Coloring Number
- Find Minimal Coloring Number: Apply Greedy backwards.
- Find Maximum Independent Set: Apply Greedy forwards.

Circle-Square Tree

- Find all 2-vertex-connected components. Build a 'square' node for them and connect it with all nodes in the same component. Then delete all edges inside the component.
- Any two 'square'('circle') node will not be adjacent.
- The shared 'circle' node of two 'square' node is the common node of the original two BCCs.
- The union of all simple paths between a and b is BCCs of all 'square' nodes on the path.
- With it, we can change most problems on graphs to trees, especially on Cactus.

Tournament Graph

- Hamilton properties has been discussed before.
- Score Sequence: Sort all nodes's out-degree in non-decreasing order.
- Landau's Theorem: A Sequence S will be considered a legal Score Sequence when for any $1 \leqslant k \leqslant n$, $C(k, 2) \leqslant \sum_{i=1}^{k} s_i$. When $k = n$, $=$ must be hold.
- More importantly: Construct the tournament graph of given sequence.