Bipartite Matching

Andrew Stankevich

December 3, 2020

1 Maximum Matching Problem

Matching is a subset $M \subset E$ of edges of a graph such that no edges in M have a common vertex. Maximum matching problem is to find a matching M that has maximum number of edges. Many problems can be reduced to maximum matching.

Given a graph G and a matching M in it, the vertex is *matched* or *covered* if it is an endpoint of an edge in M. Otherwise the vertex is *free*. *Alternating* path is the path whose edges belong alternately to the matching and not to the matching. *Augmenting* path is an alternating path that begins and ends in free vertices.

Berge's Theorem. The matching M in the graph G is maximum if and only if there is no augmenting path.

Proof. Consider a matching M and the maximum matching M_{max}. Take their symmetric difference $M \oplus M_{max}$. The resulting graph has four types of connected components:

1. balanced paths, that have the same number of edges from M and from M_{max};
2. augmenting paths for M, that have the one more edges from M_{max} than from M;
3. augmenting paths for M_{max}, that have the one more edges from M than from M_{max};
4. alternating cycles that also have the same number of edges from M and from M_{max}.

If M is not maximum, M_{max} has more edges, so there is at least one augmenting path for M.

If M is maximum there are clearly no augmenting paths, because changing edge status along such path P, removing from M edges of P and adding edges of P that didn't belong to M, we get a matching with greater number of edges. This can be also stated as $M \leftarrow M \oplus P$.

2 Bipartite Graphs

The Berge's theorem gives the algorithm of finding the maximum matching:

- Start with an empty matching M.
- Find an augmenting path P and perform XOR operation on P and M: $M \leftarrow M \oplus P$. Now the number of edges in M is increased by one.
- If there is no augmenting path, the matching M is maximum.

The problem to find augmenting path. The main obstacles are odd length cycles.

Graph G is called *bipartite* if its vertices can be divided into two parts L and R such that any edge connects vertices from different parts.

Lemma. G is bipartite if and only if it has no odd length cycles.

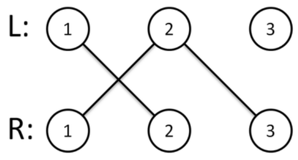
In bipartite graphs two alternating paths that start in the same vertex u can only reach some other vertex v with the same type of edge: either matching edge, or non-matching edge. This gives way to different algorithms for finding augmenting paths.

3 Algorithms

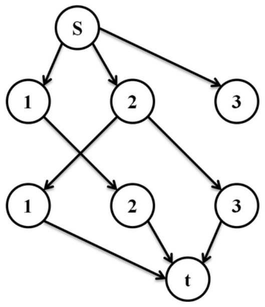
3.1 Reduction to Flows, Ford-Fulkerson Algorithm

One way to find maximum matching in bipartite graph is to reduce maximum matching problem to maximum flow problem.

Consider a bipartite graph with parts L and R.



Add two vertices s and t, add arcs from s to L, add arcs from R to t, direct edges from L to R, make capacity of all edges equal to 1.



Now the integer maximum flow in the resulting graph has easy correspondence to the maximum matching in the original bipartite graph.

Since each augmenting path increases flow by 1, flow doesn't exceed V where V is the number of vertices, one augmenting path can be found in $O(E)$ using DFS, time complexity is $O(VE)$.

3.2 Implementation details

Finding augmenting path using DFS. Let us keep two arrays: `pl[u]` is the vertex matched to u for u in L, `pr[v]` is the vertex matched to v for v in R. Let us use -1 as a marker that the vertex has no pair.

```
bool dfs(x):
    if vis[x]
        return false
    vis[x] = true

    for (y : g[x])
        if (pr[y] == -1 || dfs(pr[y])
            pr[y] = x
            pl[x] = y
            return true
    return false
```

Now one DFS from s can be implemented as a series of `dfs` calls from all free vertices of L.

```
fill(pl, -1)
fill(pr, -1)

foundPath = true
while foundPath
    foundPath = false
    fill(vis, false)
    for x in L
        if pl[x] == -1
            if dfs(x)
                foundPath = true
                break
```

Note that whole `while` iteration takes $O(E)$ because it simulates one DFS from s.

3.3 Kuhn's Algorithm

Kuhn's algorithm uses the same `dfs` from vertex as Ford-Fulkerson algorithm, but in a different way.

```
fill(pl, -1)
fill(pr, -1)

for x in L
    if pl[x] == -1
        fill(vis, false)
        dfs(x)
```

The main idea: if no augmenting path from u has been found, there would never be any augmenting path from u if we transform our matching only by augmenting paths.

Proof. Consider a set $V(u)$ of vertices reachable from u. There is no free vertex of R in $V(u)$. In order for `dfs` from u to find augmenting path the set $V(u)$ must change, but it cannot, because no augmenting path can pass through $V(u)$ since no edges go out of it. Therefore $V(u)$ will stay as it is, and there is never a reason to run `dfs` rom u again.

This fact can also be proved using matroid theory.

3.4 Hopcroft-Karp Algorithm

Consider Dinic's algorithm adapted from general graphs to graphs that are the result of bipartite matching problem reduction to maximum flow problem. This algorithm is called Hopcroft-Karp Algorithm.

One stage of Hopcroft-Karp Algorithm consists of two steps.

1. Run BFS to find distance $d[u]$ from s to u for each vertex.

2. Run Ford-Fulkerson algorithm, but

    - in DFS only go to vertices with greater value of d;
    - never go twice along the same edge.

Lemma. After each stage the distance from s to t strictly increases.

Proof. Every time the algorithm traverses an edge on the second step there are two cases: augmenting path is found using this edge, in this case the edge direction is reversed (its status in the matching is changed), so it cannot be in the shortest path any more; or there is no augmenting path using this edge, in this case it will not appear after future augmentations, because edges are only removed from the shortest path network.

Consider the shortest path from s to t before the second step of the stage. After the second step it no longer exists, because at least one edge of it has been traversed by the DFS at the second step. So no path of the same length using original edges exists.

But new edges only go backwards in shortest paths network, so no path that uses new eges can have the same or shorter length. Therefore after the second step the distance from s to t must strictly increase.

Lemma. The number of stages is at most $O(\sqrt{V})$.

Proof. Let us divide stages to *short* and *long*. We call the stage short if distance from s to t is at most $\sqrt{V}$. Clearly there are at most $\sqrt{V}$ short stages.

Consider the matching M_1 after all short stages. Let's take symmetric difference of M_1 and maximum maching M_{max}. Let $M_1 \oplus M_{max}$ have a augmenting paths for M_1, then $a = |M_{max}| - |M_1|$. Every augmenting path for M_1 has length at least $\sqrt{V}$, therefore there are at most $V/\sqrt{V} = \sqrt{V}$ such paths. So there are at most $\sqrt{V}$ long stages, and the total number of edges is at most $2\sqrt{V} = O(\sqrt{V})$ as required.

Each iteration runs in $O(E)$, and the number of iterations is $O(\sqrt{V})$. The algorithm runs in $O(E\sqrt{V})$.

3.5 Speedup Tricks

Though Kuhn's algorithm runs in $O(VE)$ its actual time on more or less random graphs is very fast. If matching is a subproblem of some larger problem and graphs look random, you can safely assume Kuhn's constant be some 1/10 to 1/20.

Greedy initialization trick helps to speed up Kuhn's algorithm even further. Before runing the algorithm, try to find greedy matching and use it as a starting point.

```
fill(pl, -1)
fill(pr, -1)

for x in L
    if pl[x] == -1
        for (y : g[x])
            if pr[y] == -1
                pl[x] = y
                pr[y] = x
                break
```

After that run Kuhn's algorithm, or any other matching algorithm.

4 Related Problems

4.1 Matchings, Covers and Independent Sets

Vertex cover is a set C of vertices such that each edge has at least one endpoint in C.

Lemma. For any matching M and any vertex cover C $|M| \leq |C|$ because each vertex can only cover one edge of M.

Corollary. The size of maximum matching is at most the size of minimum vertex cover.

Independent set is a set I of vertices such that no edge connects two vertices in I.

Lemma. C is a vertex cover if and only if $I = V \setminus C$ is an independent set.

Proof. If every edge uv has at least one endpoint in C, no edge connects two vertices in I. If no edge connects two vertices in I, each edge has at least one endpoint in C.

Clique is a set Q of vertices such that every pair of vertices in Q is connected by an edge. *Bipartite clique* B is a set of vertices in bipartite graph such that every pair of vertices from different parts is connected by an edge.

Define $\overline{G}$ for a graph G as a graph that contains an edge uv if and only if G doesn't contain such edge.

Lemma. Q is a clique in G if and only if Q is an independent set in $\overline{G}$.

Similar definition and lemma can be stated for bipartite graphs.

4.2 Kőnig's Theorem

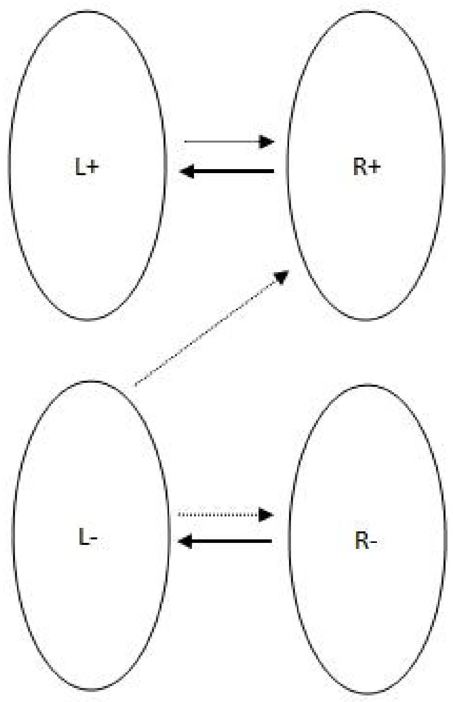
In bipartite graphs there is a more strict connection between maximum matching and minimum vertex cover.

Kőnig's Theorem. In bipartite graphs the size of the maximum matching is equal to the size of the minimum vertex cover.

Proof. We will provide the algorithm that finds the vertex cover given the maximum matching. Its size would be equal to the size of the maximum matching.

Consider the maximum matching. Run `dfs` from every free vertex of L without clearing `vis` array. Denote as L^+ all vertices of L visited by `dfs` and as L^- all vertices that were not visited. Similarly denote visited vertices of R as R^+, and non-visited vertices of R as R^-.

There are some edges between these four sets that cannot exist: by definition there are no directed edges from $+$ to $-$, also there cannot be no edges from R^- to L^+, because each vertex of L is either free (in this case there is no matching edge to it) or is visited by walking along the matching edge from R^+.



Now $C = L^- \cup R^+$ form the vertex cover and their total size is equal to the size of the matching: each matching edge has exactly one endpoint vertex in the C and no free vertex is in C.

4.3 Directed Acyclic Graphs and Path Covers

Consider a directed acyclic graph G. Create bipartite graph BG for it: split each vertex u into two vertices u^- and u^+. For each each arc uv add an edge (u^-, v^+). Now a matching in BG corresponds to the set of arcs in G such that every vertex has at most one ingoing and at most one outgoing arc. Such set of arcs forms the set of disjoint paths and cycles, but since G is acyclic, it is the set of disjoint paths.

The number of edges in paths is equal to the size of the matching M, and the number of paths is $V - M$. Therefore maximizing the size of the matching M we minimize the number of paths M.

4.4 Dilworth's Theorem

The analogue to the Kőnig's Theorem in DAG path covers terms is Dilworth's Theorem. The *antichain* in G is the set of vertices A such that no vertex from A can be reached from another vertex in A.

If we take DAG G and make its transitive closure G^*, the antichain is an independent set.

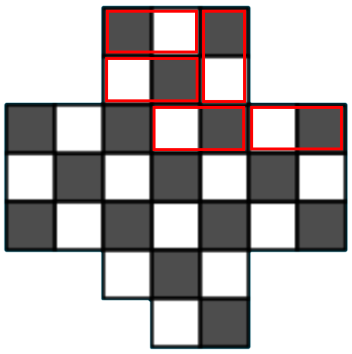
Dilworth's Theorem. The size of the maximum antichain in DAG is equal to the number of paths in the minimum paths cover in G^*.

5 Bipartite Graphs in Problems

There are some common situtation where bipartite graphs (and matchings/covers/independent sets) occur.

Two are connected with square grid figures — polyominoes.

Let grid squares be vertices of the graph. Connect two adjacent squares by an edge. A chess coloring of the grid squares shows that the graph is bipartite. Matching in such graph corresponds to *domino covering.*



Let rows and columns be vertices of the graph. Connect a row and a column if they have a common cell. Matching in such graph corresponds to *peaceful rook placement.*