Graph Matching

1. Introduction

A **Matching** in a graph **G = (V, E)** is a subset **M** of **E** edges in **G** such that no two of which meet at a common vertex.

Maximum Cardinality Matching (MCM) problem is a Graph Matching problem where we seek a matching **M** that contains the largest possible number of edges. A desirable but rarely possible result is **Perfect Matching** where all |**V**| vertices are matched (assuming |**V**| is even), i.e., the cardinality of **M** is |**V**|**/2**.

A **Bipartite Graph** is a graph whose vertices can be partitioned into two disjoint sets **U** and **V** such that every edge can only connect a vertex in **U** to a vertex in **V**.

Maximum Cardinality Bipartite Matching (MCBM) problem is the **MCM** problem in a Bipartite Graph, which is a lot easier than **MCM** problem in a General Graph.

1-1. Motivation-Applications

Graph Matching problems (and its variants) arise in various applications, e.g.,

1. The underlying reason on why Social Development Network exists in Singapore
2. Matching job openings (one disjoint set) to job applicants (the other disjoint set)
3. The weighted version of #2 is called the Assignment problem
4. Special-case of some NP-hard optimization problems (e.g., MVC, MIS, MPC on DAG, etc)
5. Sub-routine of Christofides's 1.5-approximation algorithm for TSP, etc...

1-2. Current Limitation: Unweighted Graphs

In some applications, the weights of edges are not uniform (1 unit) but varies, and we may then want to take MCBM or MCM with minimum (or even maximum) total weight.

However, this visualization is currently limited to unweighted graphs only. Thus, we currently do not support Graph Matching problem variants involving weighted graphs...

1-3. Switching Modes

To switch between the unweighted **MCBM** (default, as it is much more popular) and unweighted **MCM** mode, click the respective header.

Here is an example of **MCM** mode. In **MCM** mode, one can draw a **General**, not necessarily **Bipartite** graphs. However, the graphs are unweighted (all edges have uniform weight 1).

The available algorithms are different in the two modes.

2. Visualisation

You can view the visualisation here!

For **Bipartite Graph** visualization, we will mostly layout the vertices of the graph so that the two disjoint sets (**U** and **V**) are clearly visible as Left (**U**) and Right (**V**) sets. When you draw your input bipartite graph, you can choose to re-layout your bipartite graph into this easier-to-visualize form.

For **General Graph**, we do not (and usually cannot) relayout the vertices into this Left Set and Right Set form.

Initially, edges have grey color. Matched edges will have black color. Free/Matched edges along an augmenting path will have Orange/Light Blue colors, respectively.

3. Input Graph

There are three different sources for specifying an input graph:

1. **Draw Graph**: You can draw **any** undirected unweighted graph as the input graph.
   However, due to the way we visualize our MCBM algorithms, we need to impose one additional graph drawing constraint that does not exist in the actual MCBM problems. That constraint is that vertices on the left set are numbered from [0, n), and vertices on the right set are numbered from [n, n+m).
2. **Modeling**: Several graph problems can be reduced into an **MCBM** problem. In this visualization, we have the modeling examples for the famous Rook Attack problem and standard **MCBM** problem (also valid in **MCM** mode).
3. **Examples**: You can select from the list of our example graphs to get you started. The list of examples is slightly different in the two **MCBM** vs **MCM** modes.

4. MCBM Algorithms

There are several Max Cardinality Bipartite Matching (MCBM) algorithms in this visualization, plus one more in Max Flow visualization:

1. By reducing MCBM problem into a Max-Flow problem in polynomial time, we can actually use any Max Flow algorithm to solve MCBM.
2. O(**VE**) **Augmenting Path Algorithm** (without greedy pre-processing),
3. O(√**(V)E**) **Dinic's** or **Hopcroft-Karp Algorithm**,
4. O(**kE**) **Augmenting Path Algorithm** (with randomized greedy pre-processing),

PS1: Although possible, we will likely not use O($\mathbf{V^3}$) **Edmonds' Matching Algorithm** if the input is guaranteed to be a **Bipartite Graph** (as it is much slower).

PS2: Although possible, we will also likely not use O($\mathbf{V^3}$) **Kuhn-Munkres Algorithm** (not available in VisuAlgo yet) if the input is guaranteed to be an **unweighted** Bipartite Graph (again, as it is much slower).

4-1. MCBM ≤p Max-Flow

The **MCBM** problem can be modeled (or reduced into) as a Max Flow problem in polynomial time.

Go to Max Flow visualization page and see the flow graph modeling of MCBM problem (select Modeling → Bipartite Matching → all 1). Basically, create a super source vertex **s** that connects to all vertices in the left set and also create a super sink vertex **t** where all vertices in the right set connect to **t**. Keep all edges in the flow graph **directed** from source to sink and with unit weight 1.

If we use one of the earliest Max Flow algorihtm, i.e., a simple Ford-Fulkerson algorithm, it will still be much faster than O(**mf** × **E**) as all edge weights in the flow graph are unit weight, i.e., in O(**E^2**).

If we use one of the fastest Max Flow algorithm, i.e., Dinic's algorithm on this flow graph, we can find Max Flow = MCBM in O(√**(V)E**) time — the analysis is omitted for now. This allows us to solve MCBM problem with **V** ∈ [1000..1500] in a typical 1s allowed runtime in many programming competitions.

Discussion: The edges in the flow graph must be directed. Why?
Then prove that this reduction is correct.

4-2. Potential Issue & The Correct Reduction

[This is a hidden slide]

4-3. Should We Stop Here?

Actually, we can just stop here, i.e., when given any MCBM(-related) problem, we can simply reduce it into a Max-Flow problem and use (the fastest) Max Flow algorithm.

However, there is a far simpler Graph Matching algorithm that we will see in the next few slides. It is based on a crucial theorem and can be implemented as an easy variation of the standard Depth-First Search (DFS) algorithm.

4-4. Augmenting Path (Berge's) Theorem

Augmenting Path is a path that starts from a free (unmatched) vertex **u** in graph **G** (note that **G** does not necessarily has to be a bipartite graph although augmenting path, if any, is much easier to find in a bipartite graph), alternates through unmatched (or free/'f'), matched (or 'm'), ..., unmatched ('f') edges in **G**, until it ends at another free vertex **v**. The pattern of any Augmenting Path will be fmf...fmf and is of odd length.

If we flip the edge status along that augmenting path, i.e., fmf...fmf into mfm...mfm, we will increase the number of edges in the matching set **M** by exactly 1 unit and eliminates this augmenting path.

In 1957, Claude Berge proposes the following theorem:
*A matching **M** in graph **G** is maximum iff there is no more augmenting path in G.*

Discussion: In class, prove the correctness of Berge's theorem!
In practice, we can just use it verbatim.

4-5. Proof of Berge's Theorem

[This is a hidden slide]

4-6. M∈G is max → there is no AP in G w.r.t M

[This is a hidden slide]

4-7. M∈G is max ← there is no AP in G w.r.t M

[This is a hidden slide]

4-8. Proof, Continued (1)

[This is a hidden slide]

4-9. Proof, Continued (2)

[This is a hidden slide]

4-10. O(VE) Augmenting Path Algorithm

Recall: Berge's theorem states:
*A matching **M** in graph **G** is maximum iff there is no more augmenting path in G.*

The **Augmenting Path Algorithm** (on Bipartite Graph) is a simple O(**V*(V+E)**) = O($\mathbf{V^2 + VE}$) = O(**VE**) implementation (a modification of DFS) of that theorem: Find and then eliminate augmenting paths in Bipartite Graph **G**.

Click Augmenting Path Algorithm Demo to visualize this algorithm on a special test case called $\bar{X}$ (X-bar).

Basically, this Augmenting Path Algorithm scans through all vertices on the left set (that were initially free vertices) one by one. Suppose **L** on the left set is a free vertex, this algorithm will recursively (via modification of DFS) go to a vertex **R** on the right set:

1. If **R** is another free vertex, we have found one augmenting path (e.g., Augmenting Path 0-2 initially), and
2. If **R** is already matched (this information is stored at **match[R]**), we immediately return to the left set and recurse (e.g, path 1-2-immediately return to 0-then 0-3, to find the second Augmenting Path 1-2-0-3)

4-11. Example C++ Code - Part 1

```
vi match, vis;           // global variables

int Aug(int L) {         // notice similarities with DFS algorithm
  if (vis[L]) return 0;  // L visited, return 0
  vis[L] = 1;
  for (auto& R : AL[L])
    if ((match[R] == -1) || Aug(match[R])) { // the key modification
      match[R] = L;      // flip status
      return 1;          // found 1 matching
    }
```

```
  return 0;               // Augmenting Path is not found
}
```

4-12. Example C++ Code - Part 2

```
// in int main(), build the bipartite graph
// use directed edges from left set (of size VLeft) to right set
  int MCBM = 0;
  match.assign(V, -1);
  for (int L = 0; L < VLeft; ++L) { // try all left vertices
    vis.assign(VLeft, 0);
    MCBM += Aug(L);       // find augmenting path starting from L
  }
  printf("Found %d matchings\n", MCBM);
```

Please see the full implementation at Competitive Programming book repository: mcbm.cpp | py | java | ml.

4-13. An Extreme Test Case

If we are given a **Complete** Bipartite Graph $\mathbf{K_{N/2,N/2}}$, i.e.,
$\mathbf{V = N/2+N/2 = N}$ and $\mathbf{E = N/2 \times N/2 = N^2/4 \approx N^2}$, then
the Augmenting Path Algorithm discussed earlier will run in $O(\mathbf{VE}) = O(\mathbf{N \times N^2}) = O(\mathbf{N^3})$.

This is only OK for $\mathbf{V} \in [400..500]$ in a typical 1s allowed runtime in many programming competitions.

Try executing the **standard** Augmenting Path Algorithm on this Extreme Test Case, which is an almost complete $\mathbf{K_{5,5}}$ Bipartite Graph.

It feels bad, especially on the latter iterations...
So, should we avoid using this simple Augmenting Path algorithm?

4-14. O(√(V)E) Hopcroft-Karp Algorithm

The key idea of Hopcroft-Karp (HK) Algorithm (invented in 1973) is identical to Dinic's Max Flow Algorithm, i.e., prioritize shortest augmenting paths (in terms of number of edges used) first. That's it, augmenting paths with 1 edge are processed first before longer augmenting paths with 3 edges, 5 edges, 7 edges, etc (the length always increase by 2 due to the nature of augmenting path in a Bipartite Graph).

Hopcroft-Karp Algorithm has time complexity of $O(\sqrt{\mathbf{(V)E}})$ — analysis omitted for now. This allows us to solve MCBM problem with $\mathbf{V} \in [1000..1500]$ in a typical 1s

allowed runtime in many programming competitions — the similar range as with running Dinic's algorithm on Bipartite Matching flow graph.

Try HK Algorithm on the same Extreme Test Case earlier. You will notice that HK Algorithm can find the MCBM in a much faster time than the previous standard O(**VE**) Augmenting Path Algorithm.

Since Hopcroft-Karp algorithm is essentially also Dinic's algorithm, we treat both as 'approximately equal'.

4-15. O(kE) Augmenting Path Algorithm Plus

However, we can actually make the easy-to-code **Augmenting Path Algorithm** discussed earlier to avoid its worst case O(**VE**) behavior by doing O(**V+E**) randomized (to avoid adversary test case) greedy pre-processing *before* running the actual algorithm.

This O(**V+E**) additional pre-processing step is simple: For every vertex on the left set, match it with a *randomly chosen* unmatched neighbouring vertex on the right set. This way, we eliminate many trivial (one-edge) Augmenting Paths that consist of a free vertex **u**, an unmatched edge **(u, v)**, and a free vertex **v**.

Try Augmenting Path Algorithm Plus on the same Extreme Test Case earlier. Notice that the pre-processing step already eliminates many trivial 1-edge augmenting paths, making the actual Augmenting Path Algorithm only need to do little amount of additional work.

4-16. Another Hard Test Case

Quite often, on **randomly generated** Bipartite Graph, the randomized greedy pre-processing step has cleared most of the matchings.

However, we can construct test case like: **Example Graphs, Corner Case, Rand Greedy AP Killer** to make randomization as ineffective as possible. For every group of 4 vertices, there are 2 matchings. Random greedy processing has 50% chance of making mistake per group (but since each group has only short Augmenting Paths, the fixes are not 'long'). Try this Test Case with Multiple Components case to see for yourself.

The worst case time complexity is no longer O(**VE**) but now O(**kE**) where **k** is a small integer, much smaller than **V**, **k** can be as small as 0 and is at most **V/2** (any maximal matching, as with this case, has size of at least half of the maximum matching). In

our *empirical experiments*, we estimate **k** to be "about √(**V**)" too. This version of Augmenting Path Algorithm Plus also allows us to solve MCBM problem with **V** ∈ [1000..1500] in a typical 1s allowed runtime in many programming competitions.

4-17. So, Max Flow or AP Route?

So, when presented with an MCBM problem, which route should we take?

1. Reduce the MCBM problem into Max-Flow and use Dinic's algorithm (essentially Hopcroft-Karp algorithm) and gets **O(√(V)E)** performance guarantee but with a much longer implementation?
2. Use Augmenting Path algorithm with Randomized Greedy Processing with **O(kE)** performance with good empirical results and a much shorter implementation?

Discussion: Discuss these two routes!

4-18. Our Take (Part 1)

[This is a hidden slide]

4-19. Our Take (Part 2)

[This is a hidden slide]

4-20. Our Take (Part 3)

[This is a hidden slide]

4-21. Related Theorems

[This is a hidden slide]

5. Weighted MCBM Algorithms

Unfortunately there is no weighted MCBM algorithm (e.g., Hungarian or Kuhn-Munkres) visualization in VisuAlgo *yet*. But the plan is to have this visualization eventually.

For this section, please refer to CP4 Book 2 Chapter 9.25 and 9.27.

6. MCM Algorithms

When Graph Matching is posed on general graphs (the MCM problem), it is (much) harder to find Augmenting Path. In fact, before Jack Edmonds published his famous paper titled "Paths, Trees, and Flowers" in 1965, this MCM problem was thought to be an (NP-)hard optimization problem.

There are two Max Cardinality Matching (MCM) algorithms in this visualization:

1. O(**V^3**) **Edmonds' Matching** algorithm (without greedy pre-processing),
2. O(**V^3**) **Edmonds' Matching** algorithm (with greedy pre-processing),

6-1. O(V^3) Edmonds' Matching Algorithm

In General Graph, we may have Odd-Length cycle. Augmenting Path is not well defined in such a graph (see below), hence we cannot easily implement Claude Berge's theorem like what we did with Bipartite Graph.

Jack Edmonds call a path that starts from a free vertex **u**, alternates between free, matched, ..., free edges, and returns to the **same** free vertex **u** as a **Blossom**. This situation is only possible if we have Odd-Length cycle, i.e., in a non-Bipartite Graph. Edmonds then proposed Blossom shrinking/contraction and expansion algorithm to solve this issue.

For details on how this algorithm works, read CP4 Section 9.28.

This algorithm can be implemented in O(**V^3**).

6-2. O(V^3) Edmonds' Matching Algorithm Plus

O(V^3) Edmonds' Matching Algorithm Plus

As with the **Augmenting Path Algorithm Plus** for the MCBM problem, we can also do randomized greedy pre-processing step to eliminate as many 'trivial matchings' as possible upfront. This reduces the amount of work of **Edmonds' Matching Algorithm**, thus resulting in a faster time complexity — analysis TBA.

7. Closing Remarks

We have not added the visualizations for weighted variant of **MCBM** and **MCM** problems. They are for future work. Among the two, **weighted MCBM** will likely be added earlier than the **weighted MCM** version.

One of the possible solution for **weighted MCBM** problem is the Hungarian algorithm. This algorithm also has another name: The Kuhn-Munkres algorithm. This algorithm relies on Berge's Augmenting Path Theorem too, but it uses the theorem slightly differently.

7-1. Programming Challenges

To strengthen your understanding about these Graph Matching problem, its variations, and the multiple possible solutions, please try solving as many of these programming competition problems listed below:

1. Standard MCBM (but need a fast algorithm): Kattis - flippingcards
2. Greedy Bipartite Matching: Kattis - froshweek2
   (you do **not** need a specific MCBM algorithm for this,
   in fact, it will be too slow if you use any algorithm discussed here)
3. Special case of an NP-hard optimization problem: Kattis - TBA
4. Rather straightforward weighted MCBM: Kattis - engaging

7-2. Implementation

To tackle those programming contest problems, you are allowed to use/modify our implementation code for Augmenting Path Algorithm (with Randomized Greedy Preprocessing): mcbm.cpp | py | java | ml