perplexity

Approximate Matching in a Bipartite Pursuit-Evasion Graph

This procedure describes an algorithm to find an **approximate matching** in a bipartite graph representing coalitions of pursuers versus evaders. The main goal is to maximize the number of evaders captured by allocating the best possible pursuer coalitions to each evader, subject to coalition capabilities.

Problem Overview

- **Graph G = (U ∪ V, E):**
  - **U:** All possible size-3 coalitions of pursuers from set P, i.e., $ U = [P]^3 $.
  - **V:** Set of evaders E.
  - **E:** Edge $ e_{sj} $ exists if coalition $ P_s $ can defeat evader $ E_j $.

Algorithm Steps

1. Initial Matching (Round 1)

- **$U_1 \leftarrow P$, $V_1 \leftarrow E$**: Use all pursuers and all evaders.
- **E_1:** All valid edges between pursuers and evaders.
- **Matching:** Compute a **maximum matching M_1** using maximum network flow in $ G_1 = (U_1 ∪ V_1, E_1) $.
- **Result Sets:**
  - $ A_1 $: Pursuers involved in matching $ M_1 $.
  - $ B_1 $: Evaders matched in $ M_1 $.

2. Second Round (Improvement via Local Search)

- **$U_2 \leftarrow [P \setminus A_1]$, $V_2 \leftarrow E \setminus B_1$**: Remove pursuers from $ A_1 $ and evaders from $ B_1 $.
- **E_2:** Edges for new possible coalitions and unmatched evaders.
- **Matching:** Compute **approximate matching M_2** in $ G_2 = (U_2 ∪ V_2, E_2) $ using **local search with size-2 swaps**—a heuristic to improve the matching quickly.
- **Result Sets:**
  - $ A_2 $: Pursuers involved in $ M_2 $.
  - $ B_2 $: Evaders matched in $ M_2 $.

3. Third Round (Further Improvement)

- **$U_3 \leftarrow [P \setminus (A_1 \cup A_2)]^3$, $V_3 \leftarrow E \setminus (B_1 \cup B_2)$:** Coalitions and evaders not yet matched.
- **E_3:** Edges for remaining coalitions and evaders.
- **Matching:** Compute **approximate matching M_3** in $ G_3 = (U_3 ∪ V_3, E_3) $ again by local search with size-2 swaps.

4. Combine Results

- **Final Approximate Matching:** $ M = M_1 ∪ M_2 ∪ M_3 $

Algorithm Highlights

- **Maximum Matching by Network Flow:** Finds the largest possible set of non-overlapping matches in a bipartite graph.
- **Local Search Heuristics:** Fast, iterative improvements by considering swapping pairs to increase total matches.
- **Progressive Rounds:** Each round fixes certain pursuer-evader pairings and reduces the problem size for subsequent matches.

Key Advantages

- **Scalable:** Reduces complexity by decomposing the problem into manageable rounds.
- **Balanced:** Maximizes initial matching, then supplements with rapid local search improvements.
- **Applicability:** Effective for multi-agent pursuit-evasion problems and other coalition matching scenarios.

Example Table of Steps

Step	Description	Key Operation
Initial Matching	All pursuer coalitions vs. evaders	Maximum matching by network flow
Round 2	Remaining pursuers/evaders	Local search with size-2 swaps
Round 3	Leftover unmatched segments	Further local search size-2 swaps
Output	Union of all matchings (M1, M2, M3)	Final approximate matching

Summary

This approach constructs an effective approximation for complex bipartite matching problems involving group capabilities and constraints, such as multi-agent pursuit-evasion games. It leverages the strength of network flow algorithms for a strong initial matching, and local search for efficient improvements thereafter. The result is a robust approximation suitable for large or computationally challenging settings.

what is the complexity of this algorithm?

The **complexity of the described algorithm** is determined by the following stages:

1. **First Round (Maximum Matching)**
    - This step computes a maximum matching in a bipartite graph. The typical algorithm used is the **Hopcroft-Karp algorithm** for unweighted bipartite graphs, which runs in $O(\sqrt{V}E)$, where V is the number of vertices and E is the number of edges[1] [2] [3].
    - Since here $U_1 = [P]^3$, the number of pursuer coalitions could be as large as $O(|P|^3)$ (if all coalitions are feasible). The number of evaders is $|E|$. The edge count E_1 would depend on which coalitions can defeat which evaders but could be up to $O(|P|^3|E|)$ in the worst case.
2. **Second and Third Rounds (Approximate Matching via Local Search)**
    - Both rounds perform **local search improvements with size-2 swaps**, a heuristic for finding large matchings (not necessarily maximum). The complexity for a basic local-search matching with size-2 swaps is, in practice, much lower than maximum matching and typically **linear to the number of edges and vertices** per iteration; the number of iterations depends on the stopping criteria and graph structure[4] [2].
    - If the number of swaps or iterations is polynomial-bounded, the complexity per round is at most $O(|E|)$ per swap iteration, with a (usually small) constant factor for the number of iterations.

Overall Time Complexity

If we let:

- $n = |P|$ (number of pursuers),
- $m = |E|$ (number of evaders),

The dominating term is the first round:

- **First round:** $O(\sqrt{|P|^3 + |E|} \cdot |E_1|)$ for Hopcroft-Karp, where $|E_1| \leq |P|^3|E|$[1] [5]. For large n and moderate graph density: $O(|P|^{1.5}|E_1|)$.
- **Second and third rounds:** Each is at most linear or low-degree polynomial in $|P|^3 + |E|$ and the edge set size for the respective subgraphs[4] [2].

Summary Table

Step	Algorithm/Technique	Typical Complexity
Maximum matching	Hopcroft-Karp	$O(\sqrt{V}E)$
Local search (2-swap)	Heuristic/approximate	$O(E)$ per swap iteration

- For **dense or large graphs** (where every coalition is feasible and nearly all edges possible), the main cost is the cubic growth in the number of pursuer coalitions, making the problem **computationally expensive** even with efficient matching algorithms.

- In **practice**, local search steps are fast, but the initial matching may be the bottleneck due to coalition explosion.

Space Complexity

- **Space complexity is** $O(V + E)$, where V and E refer to the number of vertices (coalitions + evaders) and edges (possible coalition-evader pairings), respectively[6] [7].

Conclusion

- The exact time complexity is dominated by matching all size-3 pursuer coalitions to evaders, so it's typically $O(\sqrt{|P|^3 + |E|} \cdot |E_1|)$, plus local search heuristics with linear or low-degree polynomial time[5] [1] [4] [2] [3].
- **Worst-case complexity is high** due to the combinatorial growth in possible coalitions, but the use of approximation and local heuristics helps in practice on sparser or more structured graphs.

⁂

Complexity of Matching with Edge Priorities (Tiebreak on Weights)

When you require **maximum cardinality matching as the primary goal**, and only use the edge weights (priorities) to resolve ties among matchings of equal size, the algorithm essentially finds:

1. A matching with the largest number of edges (maximum cardinality).
2. Among all such matchings, one that is optimal with respect to your weight criteria (e.g., maximum or minimum total weight, or some other priority function).

Impact on Complexity

- **Unweighted Maximum Cardinality Matching:**
  - In a plain bipartite graph, this is typically solved using the Hopcroft-Karp algorithm in $O(\sqrt{V}E)$ time, where V is the number of vertices and E is the number of edges.
- **Tie-Breaking by Weights (Priority Matching):**
  - The need to break ties among same-size matchings by a secondary criterion increases the complexity.
  - According to graph theory results and efficient algorithms for **priority matching** (also called maximum priority or lex-max matching), where vertex or edge priorities are considered after cardinality, the best known complexity is $O(VE)$ in bipartite graphs[8].
  - If your prioritization is based on edge weights, and you want the highest-weight matching *among all maximum cardinality matchings*, the best known polynomial algorithm (based on extensions of the assignment/Hungarian method or Edmonds' approach) has time complexity:

    - $O(VE)$ for bipartite graphs using methods tailored for lexicographical or prioritized matching[8] [9].
    - Another approach based on maximum weight matching (not your exact question, but related) would have complexity $O(V^3)$ for the full weighted assignment problem, but that is not needed if you only break ties by weight.
  - For your three-stage coalition-rounds algorithm, the complexity for each round where priorities are considered for tie-breaking rises to $O(VE)$ per round, dominated by the size of your coalition set and evader set.

Summary Table

Problem	Typical Complexity
Maximum cardinality matching (unweighted)	$O(\sqrt{V}E)$
Max cardinality matching w/ edge priorities	$O(VE)$
Maximum weight matching	$O(V^3)$

- $V =$ number of nodes per round; $E =$ number of edges per round.

Notes

- If coalitions are large ($|P|^3$), the increased complexity due to prioritization remains polynomial, but the cubic growth in coalition count is still the main bottleneck.
- In practice, $O(VE)$ is only modestly slower than the best unweighted algorithms and is efficient for moderate-sized graphs[8] [9].

References

- Efficient algorithms for maximum priority (lex-max) matching and related problems are documented with runtime $O(VE)$ [8] [9].
- Weighted bipartite matching (assignment problem) is $O(V^3)$, but not needed unless the main goal changes from cardinality to weight[10] [11].
- Classic unweighted matching (e.g., Hopcroft-Karp) is $O(\sqrt{V}E)$ [12] [13].

In conclusion, **adding tie-breaker priorities to maximum cardinality matching increases the time complexity per round from $O(\sqrt{V}E)$ to $O(VE)$**, but the algorithm remains polynomial and practical for many applications involving prioritized coalition-evader assignments.

⁂

with respect to the answer to the first question, How do you say that the number of Pursuers is going to be P^3, when in reality the algorithm operates by choosing all single pursuers and matching them.

You're correct: **the number of pursuers is not $ |P|^3 $;** rather, for a set of pursuers $ P $, $the graph typically represents either single pursuers$(U = P $), or coalitions of size $ k $ drawn from the pursuers ($ U = \binom{|P|}{k} $).

If the algorithm operates by choosing all single pursuers and matches them to evaders—i.e.,

- **U is just the set of pursuers, $ U = P $,**
  then the bipartite graph has $ |P| $ nodes on the pursuer side, not $ |P|^3 $[14] [15].

The confusion might come from a different model seen in some pursuit-evasion research, where **U is the set of all possible coalitions of size 3 pursuers** ($ U = [P]^3 $), making $|U| = \binom{|P|}{3}$ or possibly even $ |P|^3 $ if ordered tuples are meant[16]. In that model, each node in U represents a coalition, not a single pursuer.

However, **if your matching is based on single pursuers, then:**

- The bipartite graph has $|P|$ pursuer-nodes and $|E|$ evader-nodes[14] [15].
- The algorithm's complexity is based on these set sizes (and edge count), typically $O(\sqrt{|P| + |E|}\,|E'|)$ for Hopcroft-Karp matching[15].

Summary Table

Model in Algorithm	Pursuer node set U size	Notes				
Single pursuers ($ U = P $)	$	P	$	Standard, each pursuer matched to evaders		
Coalitions of size 3 ($ U = [P]^3 $)	$	P	^3 $ or $\binom{	P	} {3}$	Each node is a group of 3 pursuers
**So: If your description is about single pursuer-evader matching, then the algorithm and graph scale with $	P	$, and my earlier answer assumed a coalition-based model, which is not applicable for the single-pursuer case. The correct complexity uses $	P	$ as the set size, not $	P	^3 $[14] [15].**

⁂

Why Advanced Matching Algorithms Are Needed for Single Pursuers

Greedy/Linear Scan Approach

A linear or greedy scan of pursuers (or evaders) may seem sufficient:

- For each pursuer, choose the first available evader that it can catch.
- Continue until no further edges are available.

However, this produces a *maximal matching*, not necessarily a maximum matching.

- A *maximal matching* means no more edges can be added, but it may not have the largest possible number of edges.
- A *maximum matching* is the largest matching possible, maximizing the total number of matched pairs.

Example

Suppose you have:

- Pursuers: {P1, P2}
- Evaders: {E1, E2}
- Edges: (P1-E1), (P2-E1), (P2-E2)

A greedy algorithm picking P1→E1 first would leave (P2, E2) unmatched even though there is actually a perfect matching: match P1→E2 and P2→E1.

Need for Algorithms Like Hopcroft-Karp

The **Hopcroft-Karp algorithm** and similar approaches systematically search for *augmenting paths*—sequences of alternating matched and unmatched edges—to improve the matching iteratively and guarantee an optimal solution[17] [18] [19].

- The complexity for maximum bipartite matching (even for single pursuers) is generally $O(\sqrt{V}E)$ using Hopcroft-Karp, where V and E are the numbers of vertices and edges.
- Simpler algorithms (like repeated DFS or the greedy approach) can miss possible better matches and do not guarantee maximum size matchings, especially in dense or complex graphs[17] [18].

Greedy Matching vs. Maximum Matching

Algorithm	Result	Time Complexity	Optimal?
Greedy (Linear Scan)	Maximal matching	$O(V+E)$	No
Hopcroft-Karp (HK)	Maximum matching	$O(\sqrt{V}E)$	Yes

- **Greedy**: Fast, but can give sub-optimal matchings.
- **Hopcroft-Karp**: Slightly more expensive, but always finds the largest possible matching.

Conclusion

A simple linear scan is fast but does not ensure the matching is as large as possible. To guarantee the *maximum cardinality matching*—even for single pursuers and evaders with arbitrary edge connections—you need algorithms like Hopcroft-Karp[18] [17] [19].

This necessity comes from the fact that the matching structure can be complex enough that a naive linear or greedy scan can easily get stuck before reaching an optimal answer.

⁂

Maximum Network Flow vs. Hopcroft-Karp Algorithm

Relationship Between the Two

- **Maximum Network Flow** is a general algorithmic framework for determining the greatest possible flow from a source to a sink in a directed graph with capacities. Classical algorithms include Ford-Fulkerson, Edmonds-Karp, and Dinic's algorithm.
- **Hopcroft-Karp Algorithm** is a specialized algorithm used to find a maximum matching in bipartite graphs. It is optimized for this specific structure and achieves a better time complexity—$O(\sqrt{V}E)$—than general-purpose maximum flow algorithms when applied to bipartite matching problems.

How Are They Connected?

- The **maximum bipartite matching problem** can be modeled as a maximum network flow problem:
  - Construct a flow network where each left-side node connects to the source, each right-side node connects to the sink, and all original edges have capacity 1.
  - The value of the maximum flow corresponds to the size of the maximum matching.
- **Hopcroft-Karp** can be viewed as a specialized implementation built upon the same core principles as Dinic's maximum flow algorithm, tailored to unit-capacity bipartite graphs. In fact, Hopcroft-Karp improves on standard augmenting path flow algorithms by processing many shortest augmenting paths in parallel, leading to its superior running time for this problem[20] [21].

Summary Table

Algorithm	Purpose	Time Complexity	Notes
Maximum Network Flow (e.g., Edmonds-Karp)	General flow problems	$O(VE^2)$	Works for any flow network

Algorithm	Purpose	Time Complexity	Notes
Hopcroft-Karp	Maximum bipartite matching	$O(\sqrt{V}E)$	Specialized, faster for bipartite graphs [20] [21]
Dinic's Algorithm	Maximum flow; related to Hopcroft-Karp	$O(V^2E)$ (general)	Hopcroft-Karp is a special case on unit-capacity bipartite graphs

Key Takeaway

- **Hopcroft-Karp is not the same as general maximum network flow,** but it is closely related—being a specialized and much faster algorithm for solving maximum matching in bipartite graphs using ideas from network flow optimization[20] [21].

⁂

Why is Hopcroft–Karp considered a maximum bipartite matching, not maximum flow

The **Hopcroft–Karp algorithm** is considered an algorithm for **maximum bipartite matching**—not maximum flow—because its goal and structure are specifically tailored to finding the largest set of non-overlapping edges (a matching) in bipartite graphs, rather than maximizing "flow" in a generic flow network[22] [23] [24].

Key points:

- **Hopcroft–Karp directly solves the bipartite matching problem.**
  Its output is a *matching* of maximum cardinality: a set of as many edges as possible such that no two share an endpoint[22] [25] [24].
- **It uses augmenting paths**: It expands the matching by finding a maximal set of shortest augmenting paths in each phase, not by incrementally pushing flow according to capacities, as in flow algorithms[22] [23] [24].
- **The connection:**
  The maximum bipartite matching problem *can* be reduced to a maximum network flow problem by creating a flow network with unit-capacity edges and looking for integer-valued flows[26] [27]. General maximum flow algorithms like Ford-Fulkerson or Edmonds-Karp can then be applied, and the result will be a matching.
  However, the Hopcroft–Karp algorithm **is specialized and optimized for bipartite matching**, giving a better worst-case complexity ($O(|E|\sqrt{|V|})$) than general maximum flow approaches[22] [23] [24].
- **Algorithmic structure:**
  Both approaches use concepts like augmenting paths. In fact, Hopcroft–Karp can be seen as a special case of Dinic's algorithm for maximum flow, restricted to the bipartite unit-capacity setting[22] [24]. But in practical and theoretical usage, Hopcroft–Karp is recognized and used for its matching-specific, not general flow, application.

In summary:
Hopcroft–Karp is called a maximum bipartite matching algorithm rather than a maximum flow algorithm because its formulation, input, and output are for matching in bipartite graphs, and it achieves this goal much more efficiently than if one used a general maximum flow approach[22] [23] [24]. The fact that matching can be *modeled as* a flow problem does not change the fact that Hopcroft–Karp is a dedicated matching algorithm.

⁂

why does this psuedocode use maximum flow instead of hoppcroft-karp?

The pseudocode uses **maximum flow** rather than the Hopcroft–Karp algorithm because, while both approaches are mathematically equivalent for solving maximum bipartite matching, the **maximum flow framework is more general and flexible** for algorithmic design—especially when you might need to:

- **Handle priorities, weights, or additional constraints:** Maximum flow reductions can accommodate prioritization, weights, or capacity constraints more directly than the basic Hopcroft–Karp method[28] [29] [30].
- **Unify approach for different matching problems:** In more complex or extended problems (such as matching with node/edge capacities, multiple rounds, or variable coalition sizes), converting the problem to a flow network and using generic max-flow solvers brings consistency and makes it easier to adapt the algorithm to changing requirements or extra features[29] [30] [31].
- **Algorithmic familiarity and code reuse:** In practice, many algorithm libraries support robust, fast max-flow implementations (Ford-Fulkerson, Edmonds-Karp, Dinic's), and using these can simplify development, especially for researchers who might be familiar with flow-based methods.

Both approaches fundamentally find the maximum matching in a bipartite graph, but:

- **Hopcroft–Karp is specialized for unweighted bipartite matching**, running in $O(|E|\sqrt{|V|})$ [28] [32] [33].
- **Maximum flow algorithms** are more general (applicable to many problems), and for bipartite graphs with unit capacities, will also return a matching. Their time complexity is usually a bit worse, often $O(VE)$ [30] [29].

Direct answer:
The pseudocode likely uses maximum flow instead of Hopcroft–Karp **for generality and flexibility**. In particular, if future extensions to the algorithm require weights, priorities, or more complicated constraints, the maximum flow model is ready-made to handle these, even if Hopcroft–Karp is theoretically faster for the classic matching case[28] [29] [30] [31].

References:

- [28] Wikipedia: Hopcroft–Karp Algorithm

- [29] Mastering Maximum Bipartite Matching
- [30] JHU Algorithms Lecture: Max-Flow II (Bipartite Matching)
- [31] Duke Algorithms Lecture: Bipartite Matching, Max Flow

⁂

Implementing Maximum Flow for Maximum Cardinality Matching with Edge Priorities

To find a **maximum cardinality matching in a bipartite graph** while respecting a given edge priority (i.e., among all maximum matchings, prefer those with higher-priority edges), you can use **minimum-cost maximum flow (min-cost max-flow)** techniques:

Construction Steps

1. **Augment the Graph to a Flow Network**
   - Add a source node `s` and a sink node `t` to your bipartite graph.
2. **Connect Source to Left Nodes (U)**
   - For each node in the left part (`U`), add an edge from `s` to that node with *capacity 1* and *cost 0*.
3. **Connect Right Nodes (V) to Sink**
   - For each node in the right part (`V`), add an edge from that node to `t` with *capacity 1* and *cost 0*.
4. **Encode Priority in Edge Costs**
   - For every possible matching edge (`u` in `U`, `v` in `V`), add an edge from `u` to `v` with:
     - *Capacity 1*
     - *Cost reflecting the priority (e.g., use a lower cost for higher priority so that minimizing total cost selects higher priority edges first)*[34].

Solving

5. **Run a Min-Cost Max-Flow Algorithm**
   - Use a standard min-cost max-flow algorithm such as:
     - Successive Shortest Augmenting Path (SSAP)
     - Cycle Canceling
     - Capacity-Scaling Min-Cost Flow
   - These algorithms find the largest possible matching (max flow) and, among all such matchings, one with the minimum sum of edge costs (thus obeying priorities)[34].
6. **Extract the Matching**
   - After running the algorithm, the set of edges from `U` to `V` carrying a flow of 1 forms your maximum cardinality, priority-respecting matching.

Why This Works

- **Integrality:** Since all capacities are 1, the resulting flow will be 0 or 1 on each edge, exactly corresponding to the matching.
- **Priority Guarantee:** By converting priorities to edge costs, the min-cost max-flow ensures high-priority edges are selected if possible, without sacrificing the size of the matching[34].

Example

Suppose you have edge (A,1) with priority 1 and (A,2) with priority 2 (where lower number is higher priority):

- Set cost of (A,1): 1
- Set cost of (A,2): 2

The min-cost max-flow will favor using (A,1) in matching as long as it does not reduce the total number of matches.

References

- Using costs for priorities and min-cost-flow for maximum matching: [34]
- Detailed reduction of bipartite matching to flow problems: [35]

This approach is standard for handling maximum cardinality matchings with priorities or costs in bipartite graphs, and is widely used in assignment and scheduling algorithms[34] [35].

⁂

1. https://en.wikipedia.org/wiki/Matching_(graph_theory)
2. https://drops.dagstuhl.de/storage/00lipics/lipics-vol173-esa2020/LIPIcs.ESA.2020.76/LIPIcs.ESA.2020.76.pdf
3. https://www3.nd.edu/~kogge/courses/cse60742-Fall2018/Public/StudentWork/KernelPaper-v2/KernelChapter_BipartiteMatchingv_2.pdf
4. https://perso.ens-lyon.fr/bora.ucar/papers/RR-9321.pdf
5. https://cp-algorithms.com/graph/kuhn_maximum_bipartite_matching.html
6. https://www.geeksforgeeks.org/dsa/maximum-bipartite-matching/
7. https://arxiv.org/pdf/2112.14030v3.pdf
8. https://en.wikipedia.org/wiki/Priority_matching
9. https://iss.oden.utexas.edu/?p=projects%2Fgalois%2Fanalytics%2Fmax_card_bipartite_matching
10. http://www.arxiv.org/pdf/2502.20889.pdf
11. https://arxiv.org/html/2502.20889v1
12. https://cp-algorithms.com/graph/kuhn_maximum_bipartite_matching.html
13. https://math.mit.edu/~goemans/18433S15/matching-notes.pdf
14. https://taylorandfrancis.com/knowledge/Engineering_and_technology/Engineering_support_and_special_topics/Maximum_matching/
15. https://cp-algorithms.com/graph/kuhn_maximum_bipartite_matching.html

16. https://arxiv.org/pdf/1909.11881.pdf
17. https://cp-algorithms.com/graph/kuhn_maximum_bipartite_matching.html
18. https://www.geeksforgeeks.org/dsa/maximum-bipartite-matching/
19. https://www.cs.cmu.edu/~ckingsf/bioinfo-lectures/matching.pdf
20. https://en.wikipedia.org/wiki/Hopcroft–Karp_algorithm
21. https://www.cs.cornell.edu/courses/cs6820/2020fa/handouts/dinitz.pdf
22. https://en.wikipedia.org/wiki/Hopcroft–Karp_algorithm
23. https://brilliant.org/wiki/hopcroft-karp/
24. https://en.wikipedia.org/wiki/Hopcroft-Karp_algorithm
25. https://networkx.org/documentation/stable/reference/algorithms/generated/networkx.algorithms.bipartite.matching.hopcroft_karp_matching.html
26. https://www.geeksforgeeks.org/dsa/maximum-bipartite-matching/
27. http://www.columbia.edu/~cs2035/courses/ieor8100.F12/lec5.pdf
28. https://en.wikipedia.org/wiki/Hopcroft–Karp_algorithm
29. https://www.numberanalytics.com/blog/ultimate-guide-maximum-bipartite-matching
30. https://www.cs.jhu.edu/~mdinitz/classes/IntroAlgorithms/Fall2021/Lectures/Lecture19/lecture19.pdf
31. https://courses.cs.duke.edu/spring20/compsci330/lecture17scribe.pdf
32. https://brilliant.org/wiki/hopcroft-karp/
33. https://arxiv.org/abs/2312.12584
34. https://cs.stackexchange.com/questions/82553/max-flow-with-priorities
35. https://en.wikipedia.org/wiki/Maximum_cardinality_matching