香港電腦奧林匹克競賽
Hong Kong Olympiad in Informatics

Flow and Graph Matching (II)

Darren Lu {firewater}

2026-04-25

From last time…

- Make sure you have attempted the practice tasks on HKOJ for the Flow (I) lecture!

☆ A410	Maximum Flow ✓	14	Submit Submissions
☆ A411	Maximum Flow: Disjoint Paths ✓	5	Submit Submissions
☆ A412	Maximum Flow: Exam Assembly ✓	3	Submit Submissions
☆ A413	Minimum Cut ✓	4	Submit Submissions
☆ A414	Minimum Cut: Maximum Weight Closure ✓	3	Submit Submissions
☆ A415	Minimum Cut: Project Selection ✓	3	Submit Submissions

- Note: Solving A410 and A413 are **prerequisites** for this lecture.

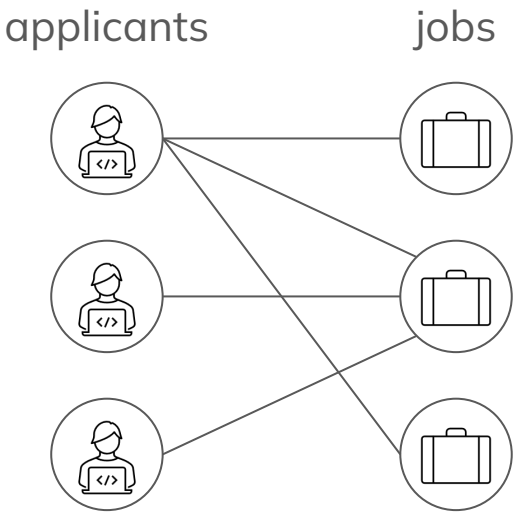
Agenda

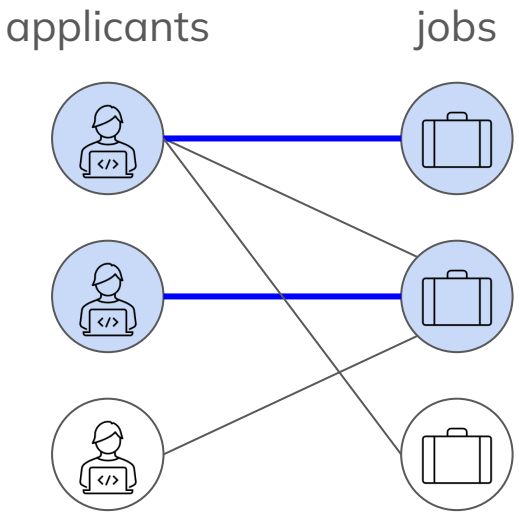
- Bipartite Matching
  - Maximum Matching
  - Minimum Vertex Cover and Maximum Independent Set
  - Applications
  - Perfect Matching and Hall's Marriage Theorem
- Flows with Demands
- *Team* Problem Set

Bipartite Matching

Bipartite Matching Problem

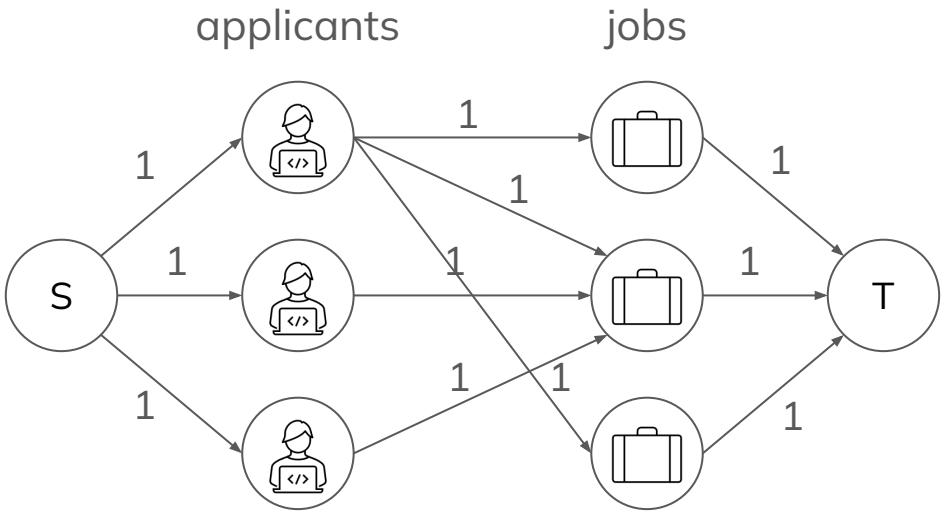
You are given a **bipartite graph**. Pick the maximum number of edges such that each vertex is incident to at most one edge.





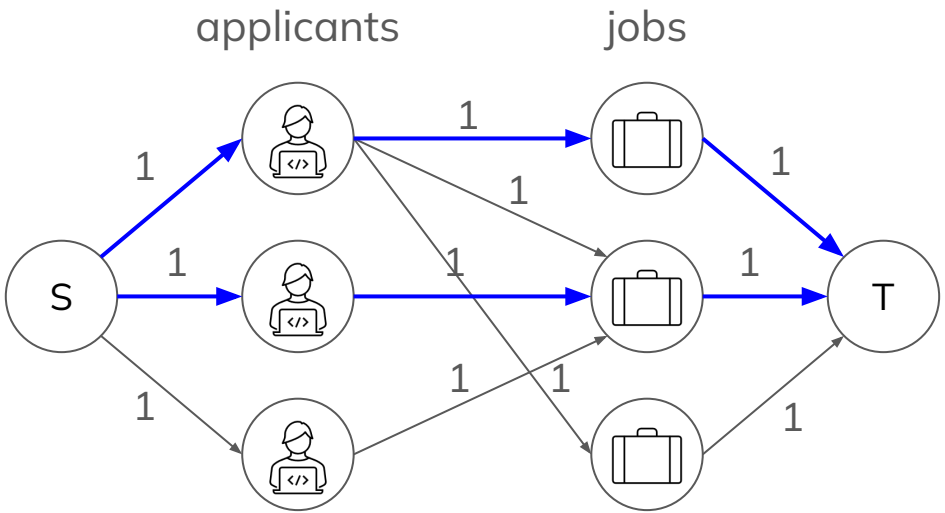
Finding Maximum Matching

The bipartite matching problem can be easily solved using **maximum flow**.



Finding Maximum Matching

The bipartite matching problem can be easily solved using **maximum flow**.



Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

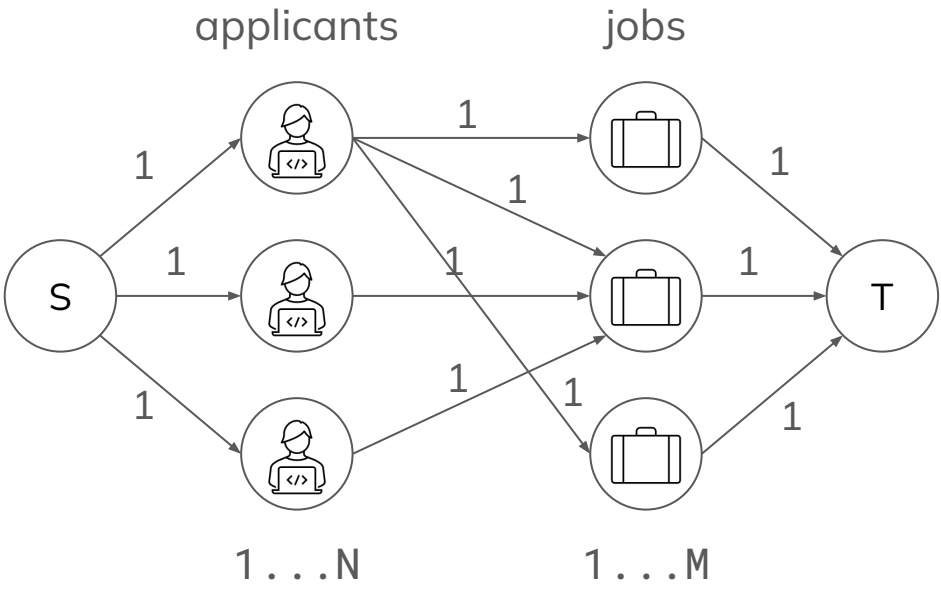
High-level idea:

- Iterate over each left vertex u (from 1 to N). For each left vertex u, iterate over all the edges incident to u.
- For such a right vertex v s.t. (u → v) exists, if it hasn't been matched by any previously considered left vertex, then **directly match u to v**.
- Otherwise, attempt to have the left vertex that was originally matched with v **find another right vertex**. If that original partner can be matched to another vertex, then match u to v; otherwise, u remains unmatched.

Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
          (find_match(match[job])))
            Match applicant to job
            Return True
    }
    Return False
}
```



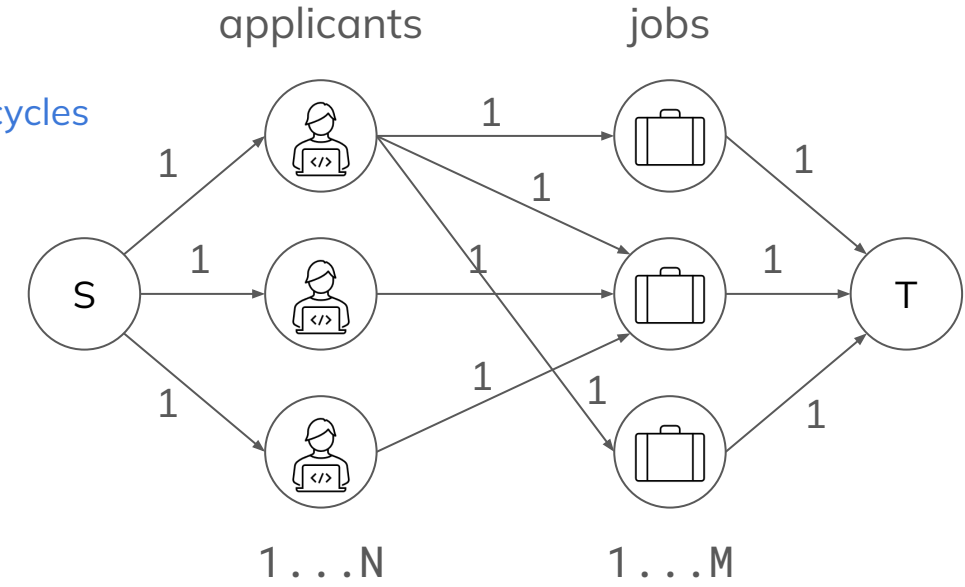
Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
            (find_match(match[job])))
                Match applicant to job
                Return True
    }
    Return False
}
```

Avoid cycles

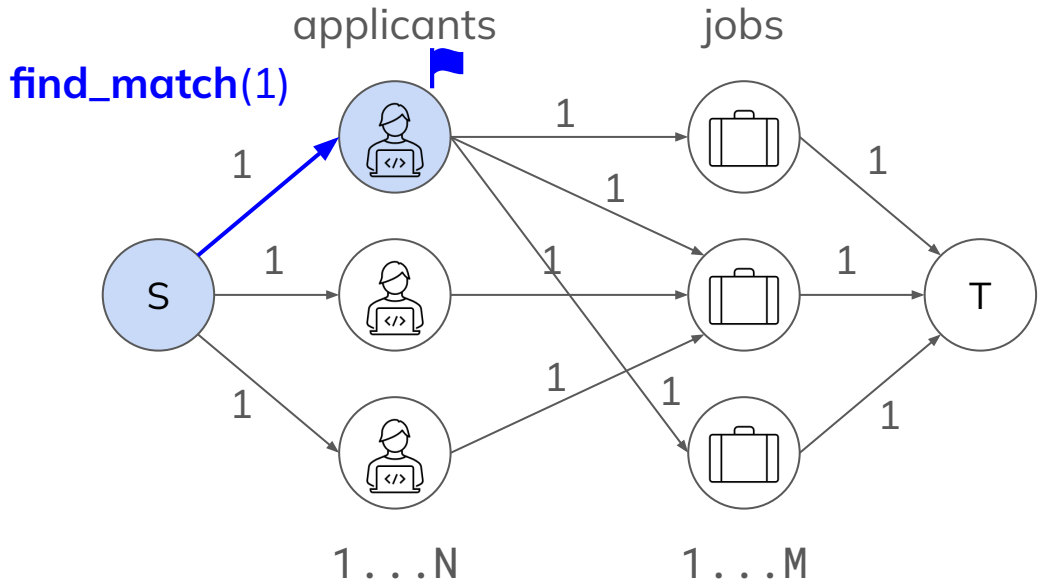
"Undo" action



Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

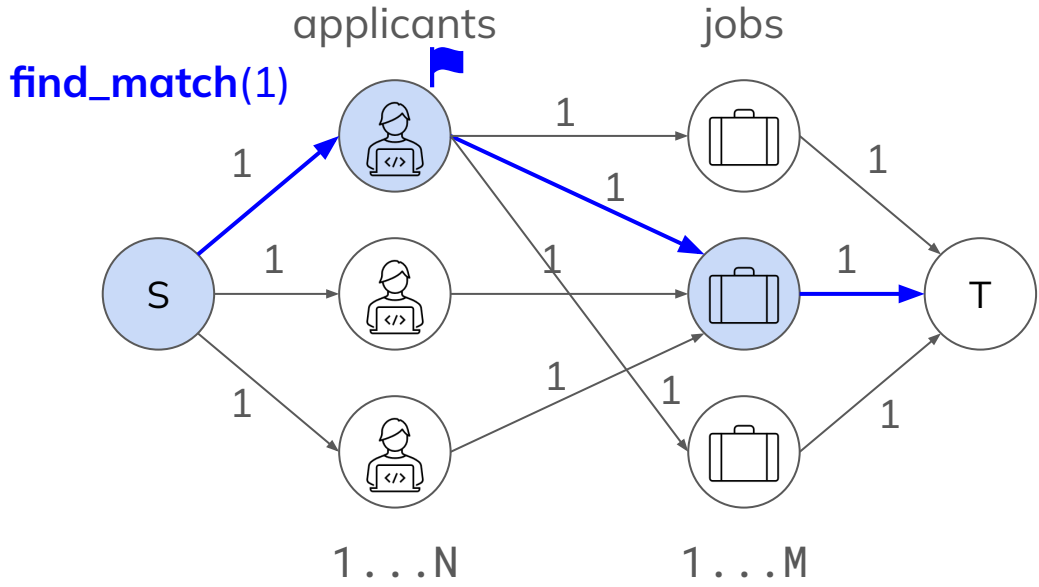
```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
          (find_match(match[job])))
            Match applicant to job
            Return True
    }
    Return False
}
```



Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

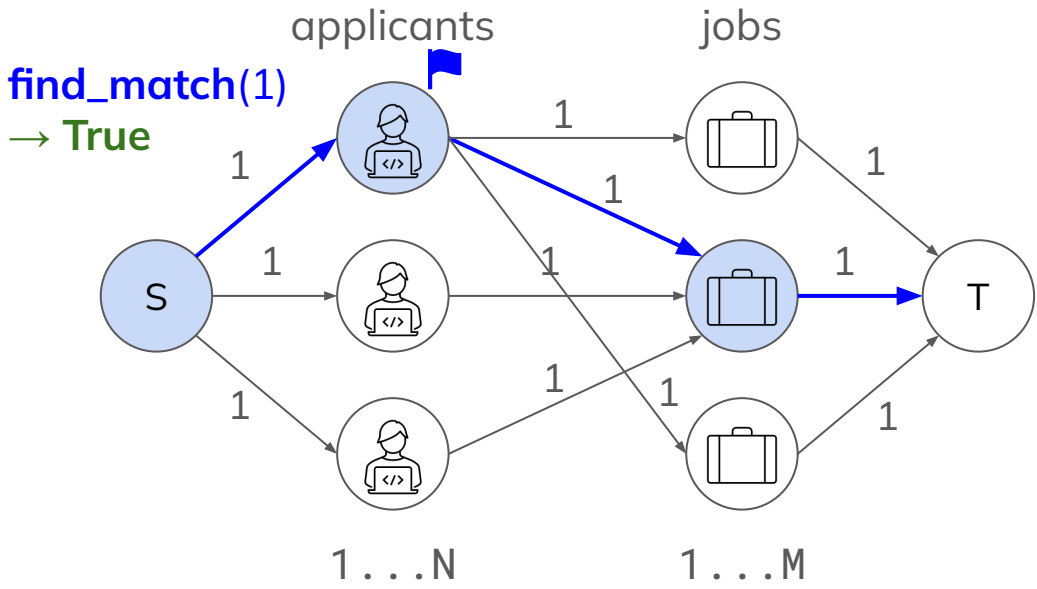
```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
          (find_match(match[job])))
            Match applicant to job
            Return True
    }
    Return False
}
```



Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

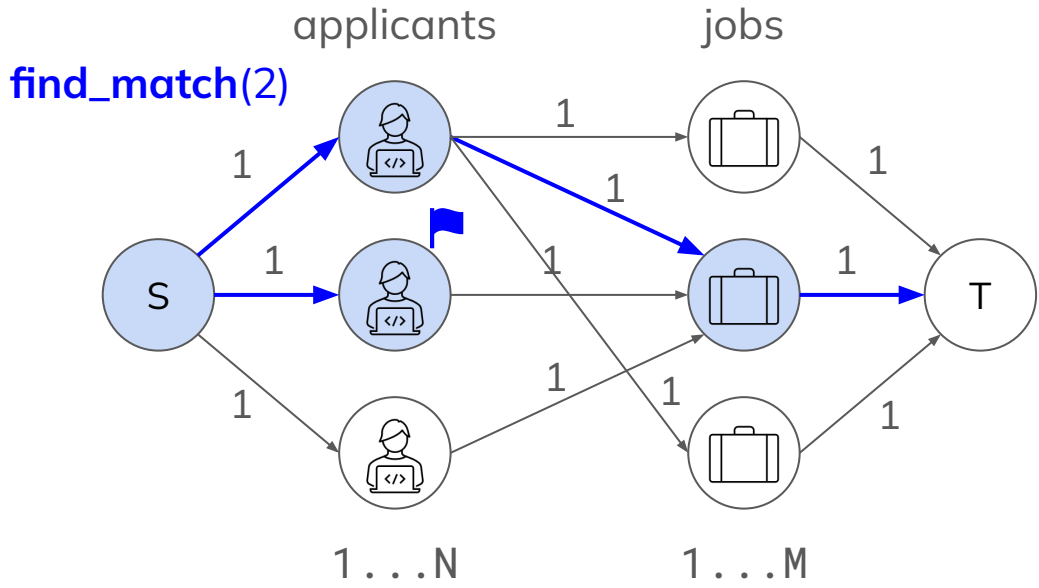
```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
          (find_match(match[job])))
            Match applicant to job
            Return True
    }
    Return False
}
```



Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

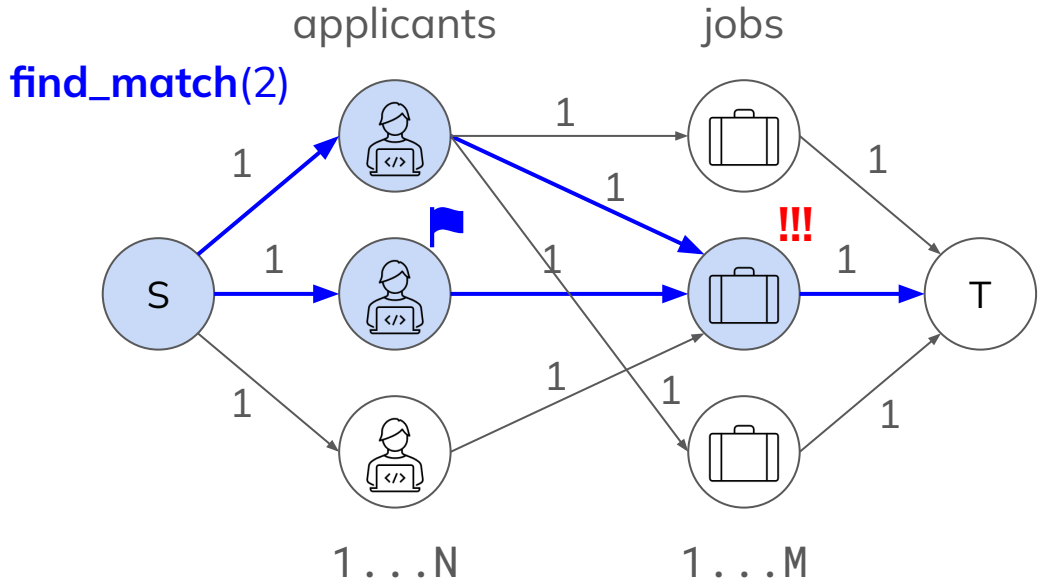
```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
          (find_match(match[job])))
            Match applicant to job
            Return True
    }
    Return False
}
```



Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

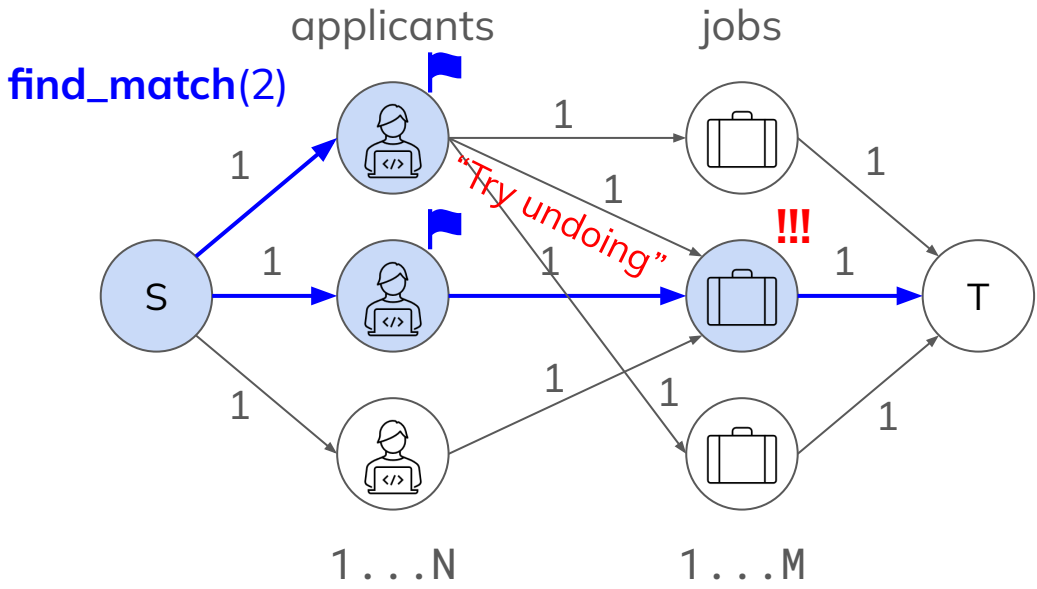
```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
          (find_match(match[job])))
            Match applicant to job
            Return True
    }
    Return False
}
```



Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

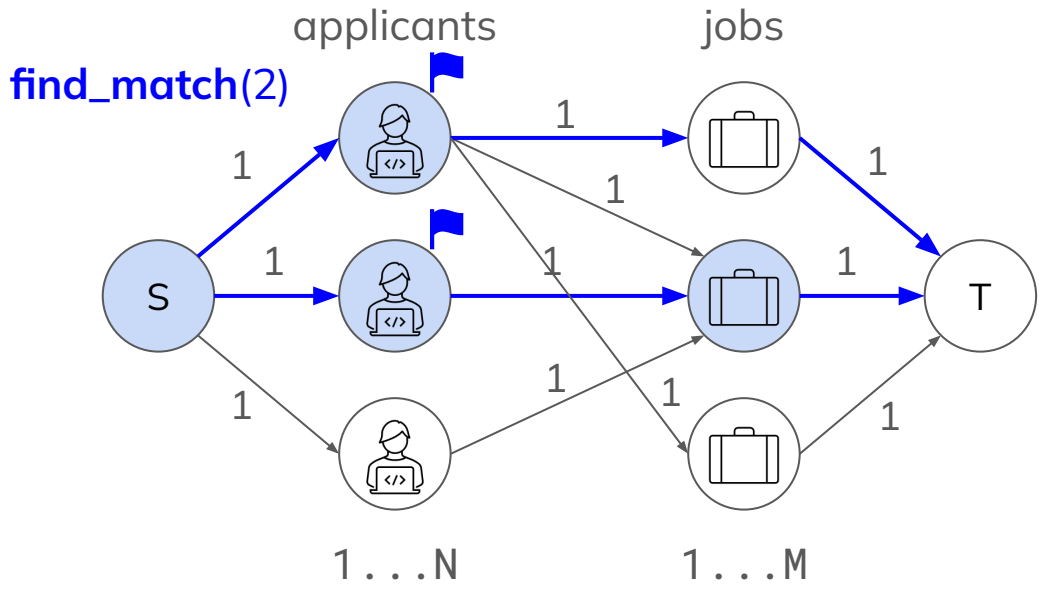
```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
          (find_match(match[job])))
            Match applicant to job
            Return True
    }
    Return False
}
```



Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

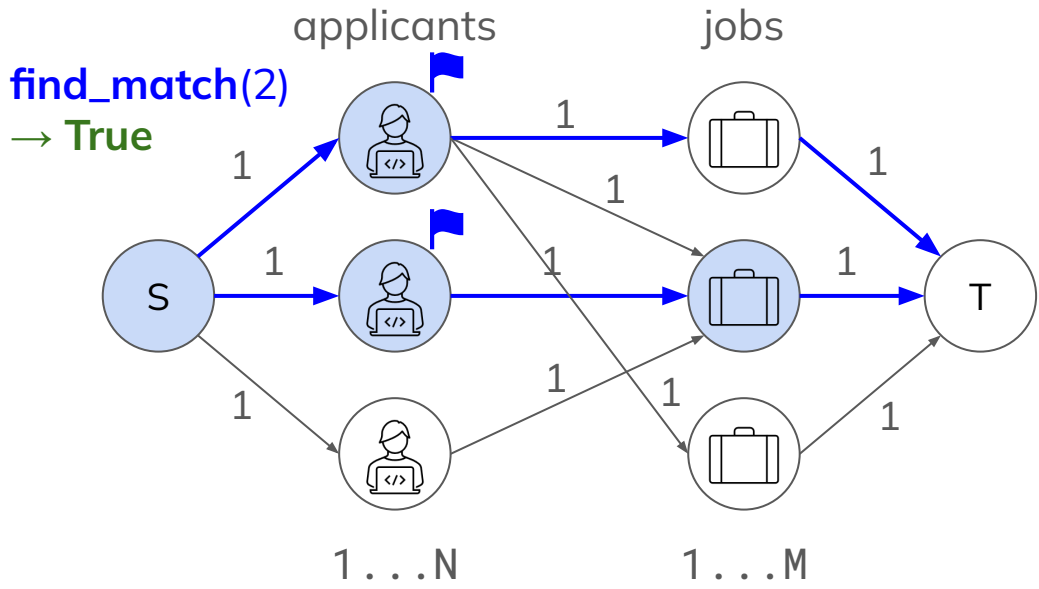
```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
          (find_match(match[job])))
            Match applicant to job
            Return True
    }
    Return False
}
```



Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

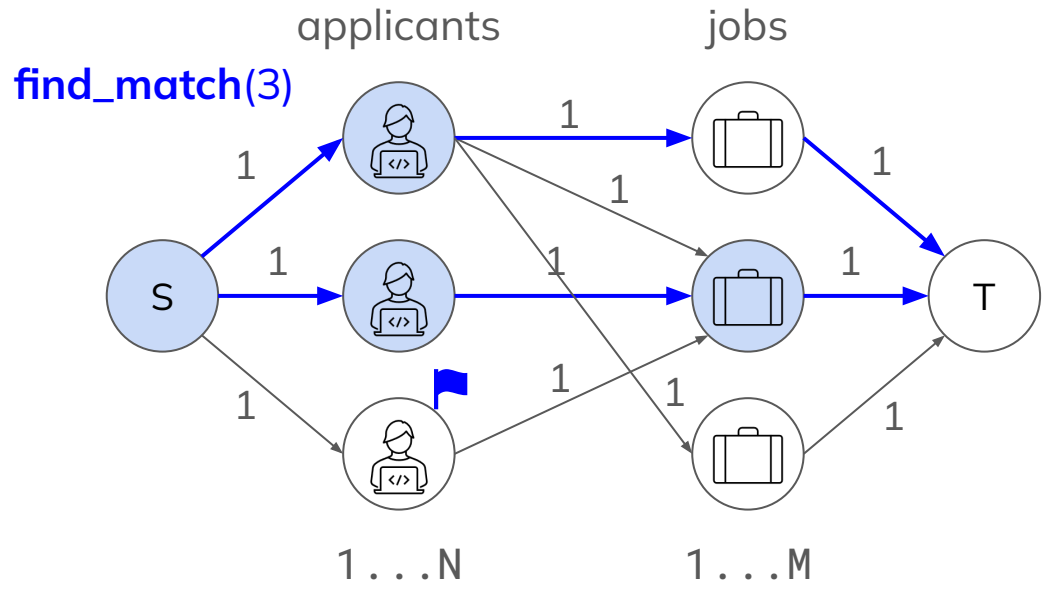
```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
          (find_match(match[job])))
            Match applicant to job
            Return True
    }
    Return False
}
```



Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

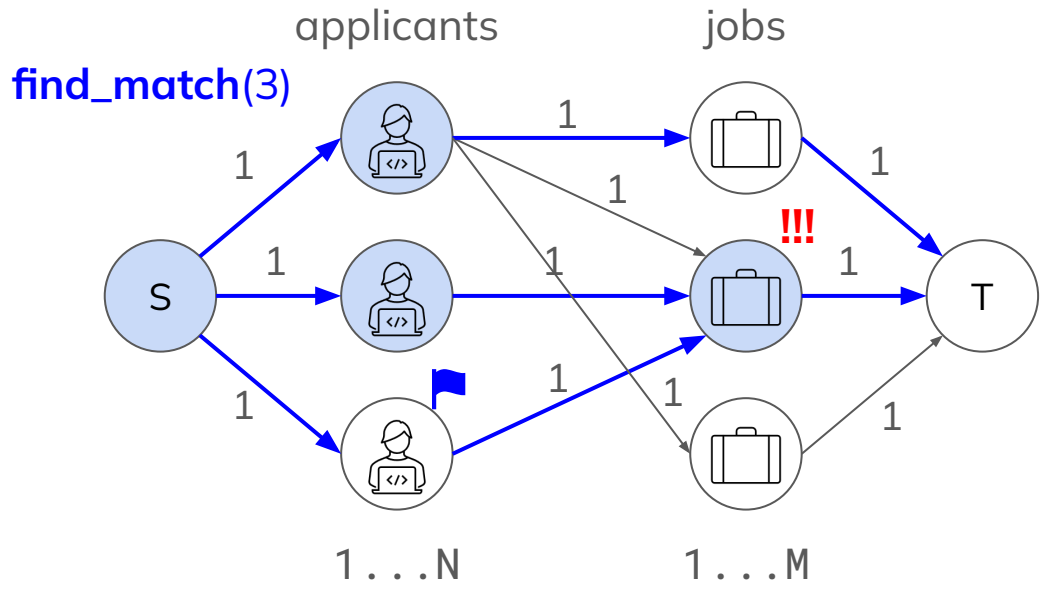
```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
          (find_match(match[job])))
            Match applicant to job
            Return True
    }
    Return False
}
```



Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

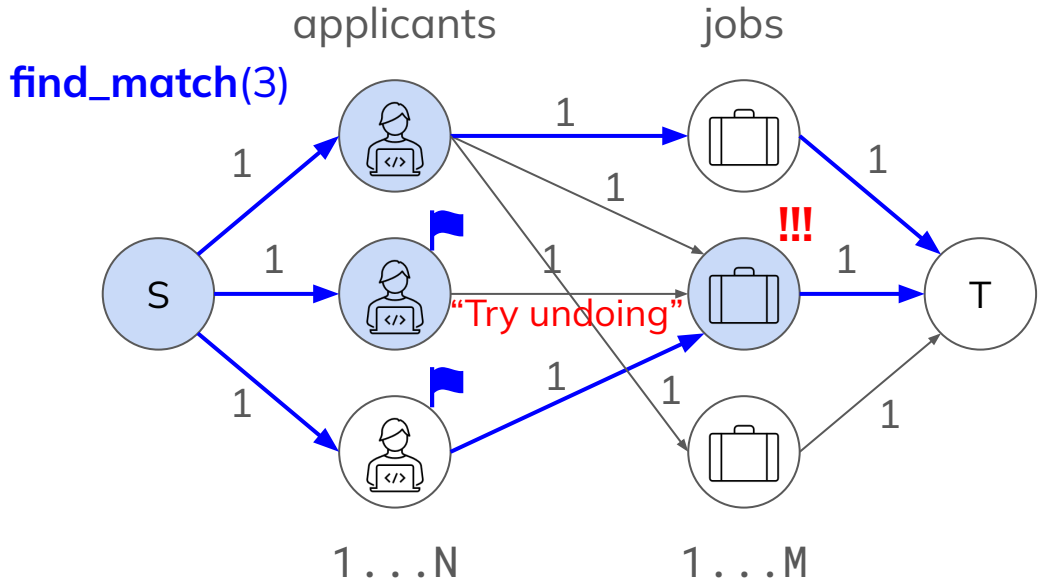
```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
          (find_match(match[job])))
            Match applicant to job
            Return True
    }
    Return False
}
```



Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

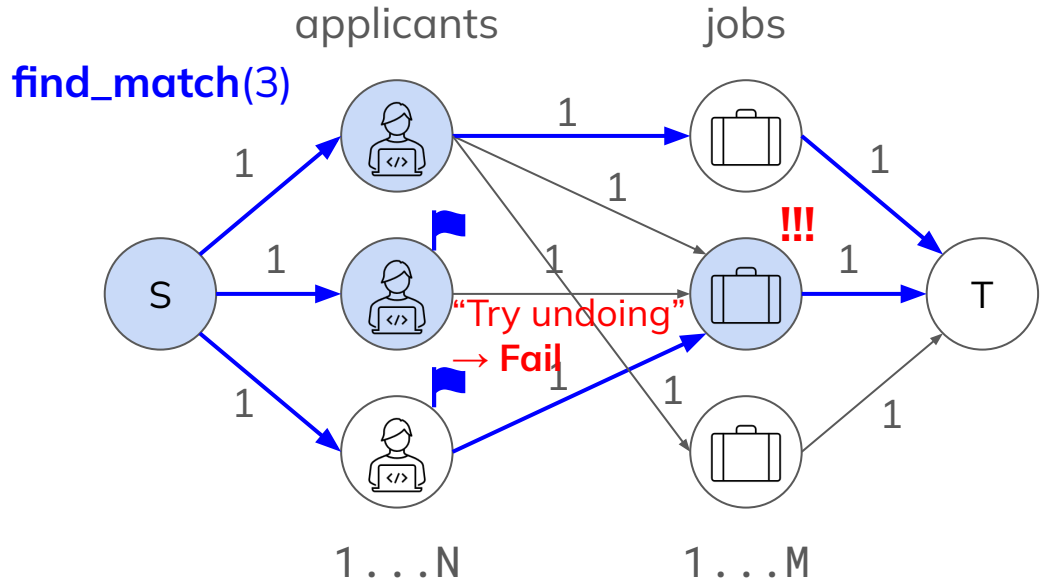
```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
          (find_match(match[job])))
            Match applicant to job
            Return True
    }
    Return False
}
```



Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

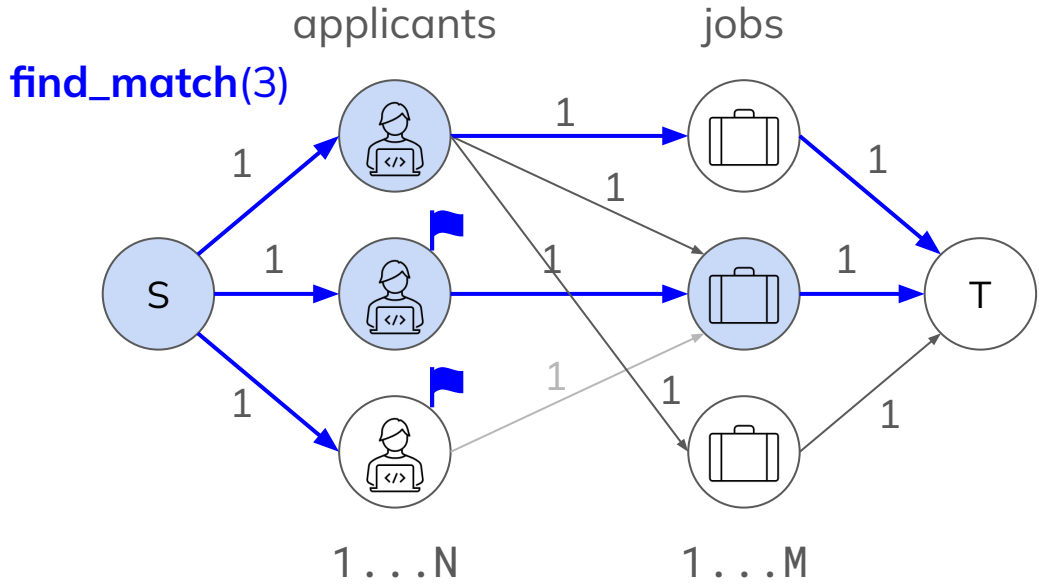
```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
          (find_match(match[job])))
            Match applicant to job
            Return True
    }
    Return False
}
```



Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

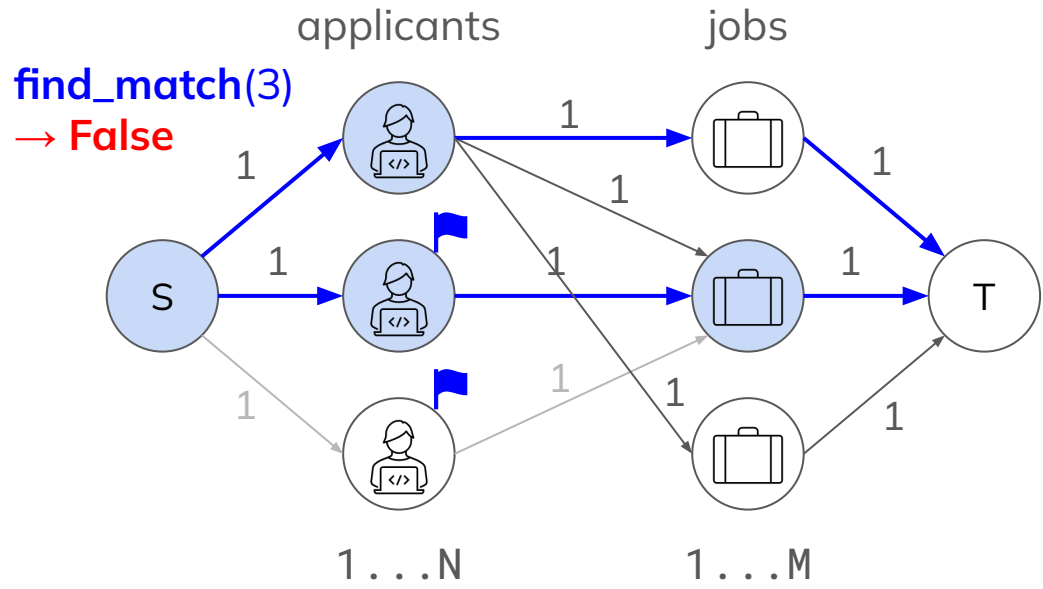
```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
          (find_match(match[job])))
            Match applicant to job
            Return True
    }
    Return False
}
```



Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

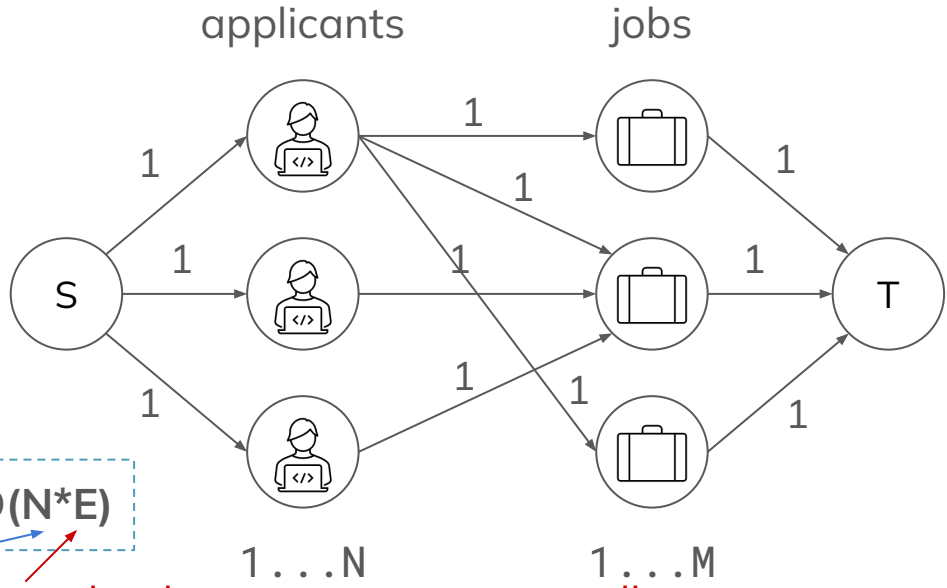
```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
          (find_match(match[job])))
            Match applicant to job
            Return True
    }
    Return False
}
```



Finding Maximum Matching – Implementation

Fortunately, we can exploit the structure of the graph to implement this more easily.

```
def find_match(applicant) {
    if (visited[applicant] is True)
        Return False
    visited[applicant] ← True
    for each (matchable job) {
        if ((job is unmatched) or
          (find_match(match[job])))
            Match applicant to job
            Return True
    }
    Return False
}
```



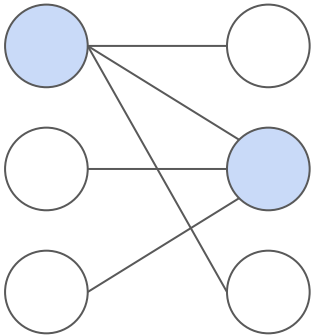
Time Complexity: **O(N*E)**

N calls

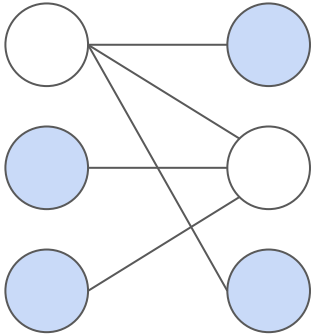
Iterate each edge at most once per call

Minimum Vertex Cover and Maximum Independent Set

Minimum Vertex Cover: Pick the **min.** number of vertices such that for each edge u → v, **at least** one of u or v is picked.



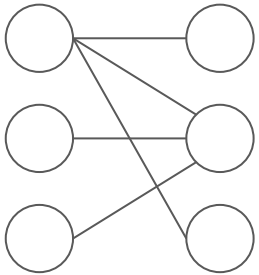
Maximum Independent Set: Pick the **max.** number of vertices such that for each edge u → v, **at most** one of u or v is picked.



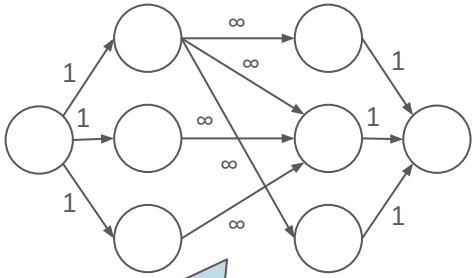
Both problems are **NP-hard** in general graphs, but **easily solved in bipartite graphs**!

Minimum Vertex Cover and Maximum Independent Set

Bipartite Matching



Maximum Flow

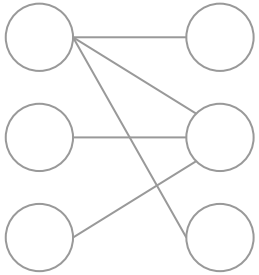


Flow does not exceed 1
→ OK to change to ∞.

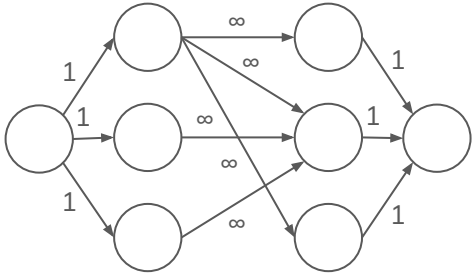
Minimum Vertex Cover and Maximum Independent Set

Max Flow = Min Cut

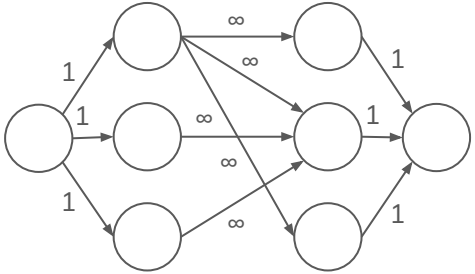
Bipartite Matching



Maximum Flow

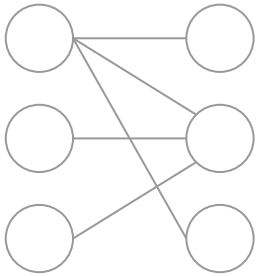


Minimum Cut

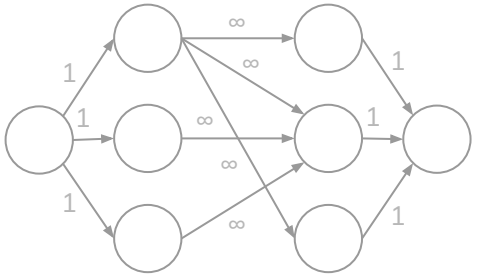


Minimum Vertex Cover and Maximum Independent Set

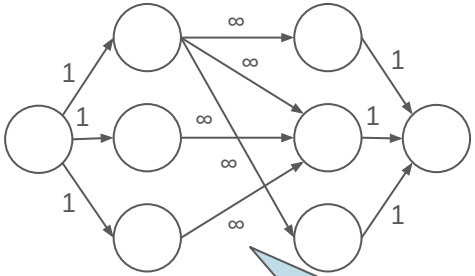
Bipartite Matching



Maximum Flow

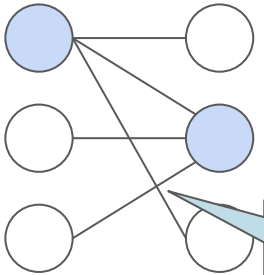


Minimum Cut



Conflict: We must pay cost 1 to cut either u or v.

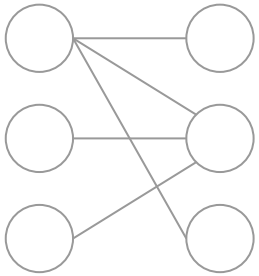
Minimum Vertex Cover



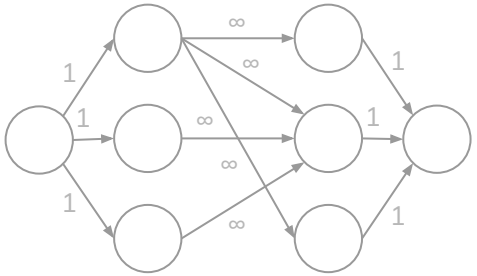
Conflict: We must pick either u or v.

Minimum Vertex Cover and Maximum Independent Set

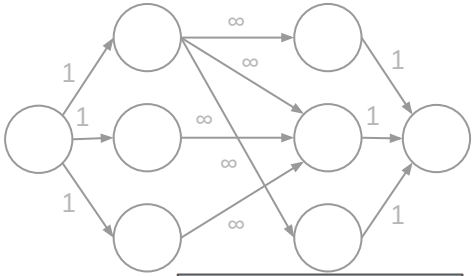
Bipartite Matching



Maximum Flow

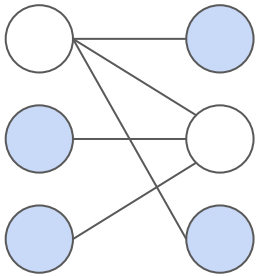


Minimum Cut



Maximum Independent Set

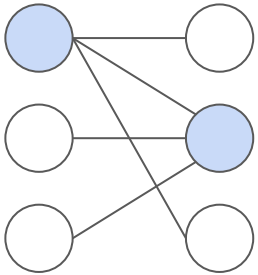
Maximize
Pick **at most**
one of u or v.



Flip the vertices

(Why?)

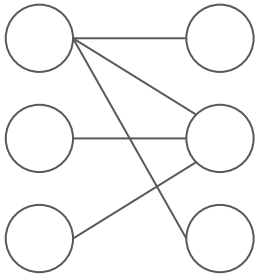
Minimum Vertex Cover



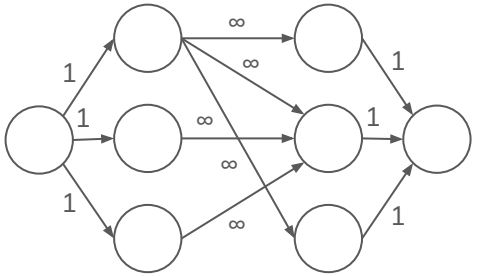
Minimize
Pick **at least**
one of u or v.

Minimum Vertex Cover and Maximum Independent Set

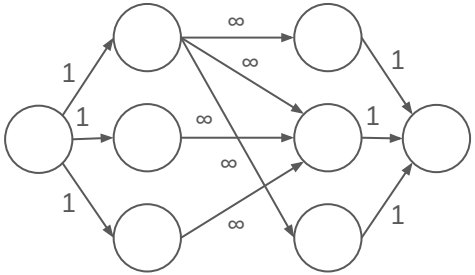
Bipartite Matching



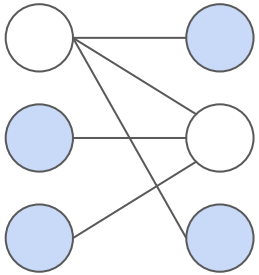
Maximum Flow



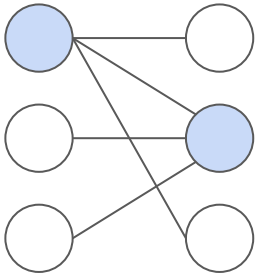
Minimum Cut



Maximum Independent Set



Minimum Vertex Cover



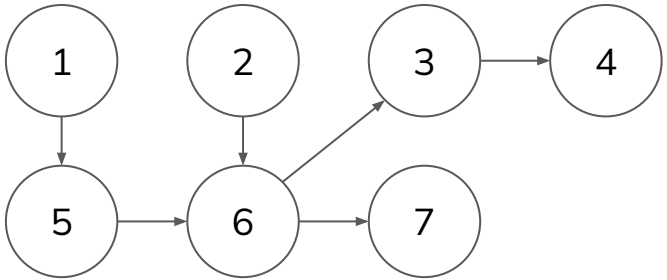
Flip the vertices

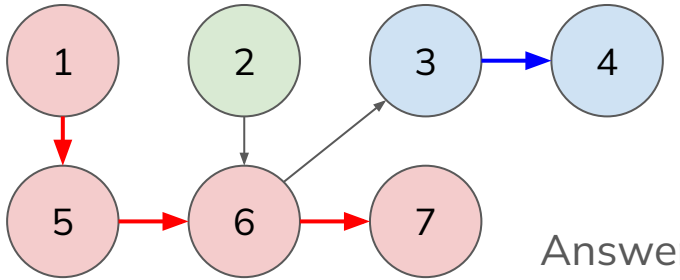
(Why?)

All problems are interrelated!

Modeling Problems with Matching

Vertex-Disjoint Path Cover: Given a directed graph. Find the minimum number of vertex-disjoint paths that each vertex belongs to exactly one path.



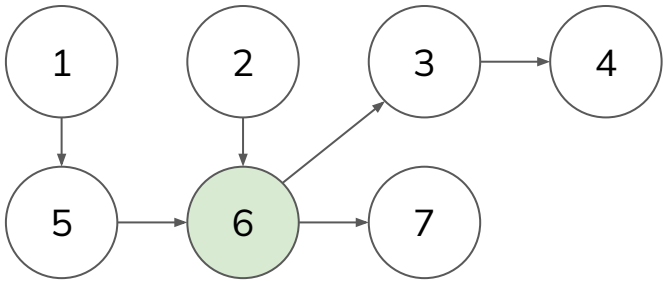


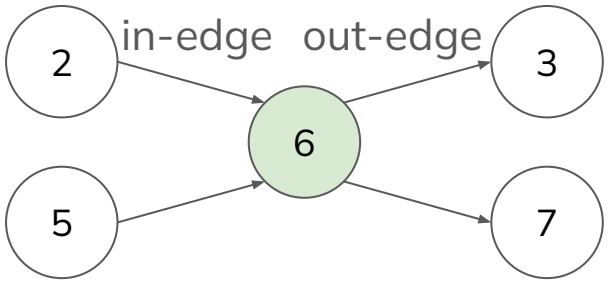
Answer: 3

Modeling Problems with Matching

For many maximum matching problems, it's useful to think in this way:

- If there are no edges, each vertex will be form its own path → answer is $|V|$.
- Each edge can "match" two vertices together → answer reduces.
  - Therefore, we want the maximum matching.

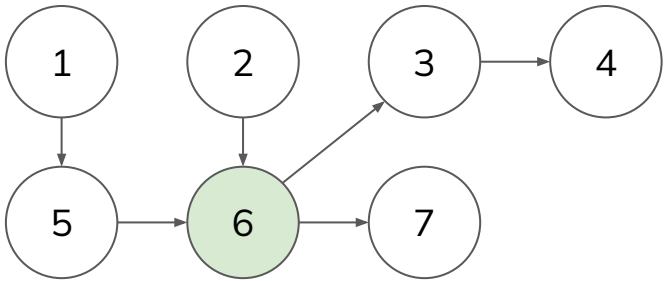


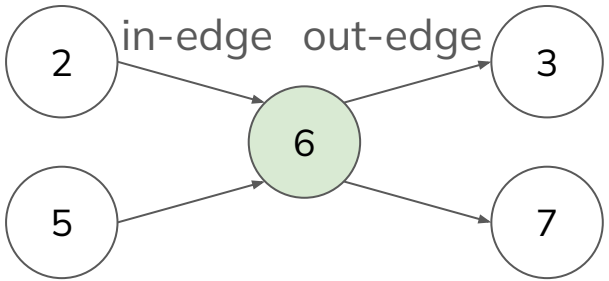


Modeling Problems with Matching

What is being “matched”?

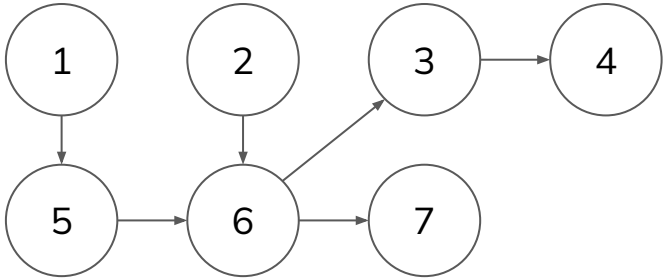
- We’re trying to match an in-edge and an out-edge for each vertex.
- We want to match as many pairs as possible → each pair reduces the answer by 1.

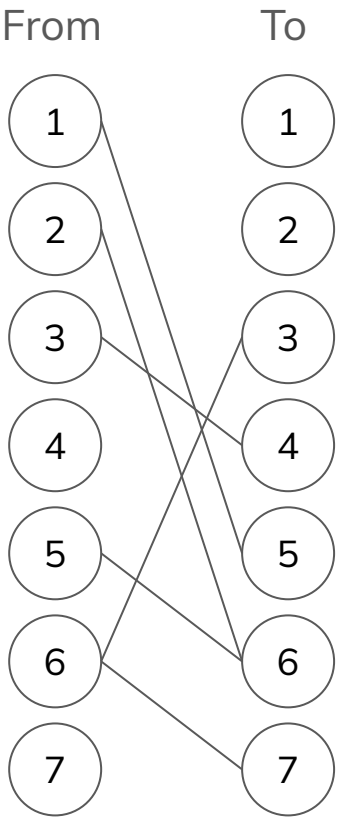




Modeling Problems with Matching

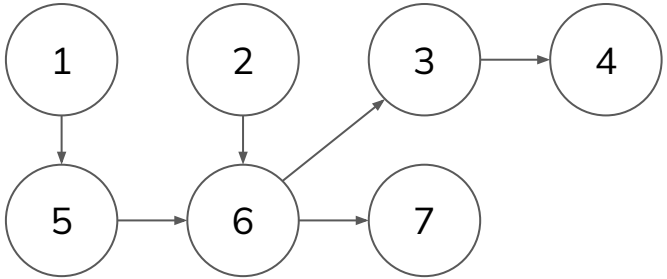
- Create a “from” vertex (out-edge) and “to” vertex (in-edge) for each vertex.
- For each edge u → v, build an edge `u.from` → `v.to`.
- Answer = `|V|` - maximum matching.

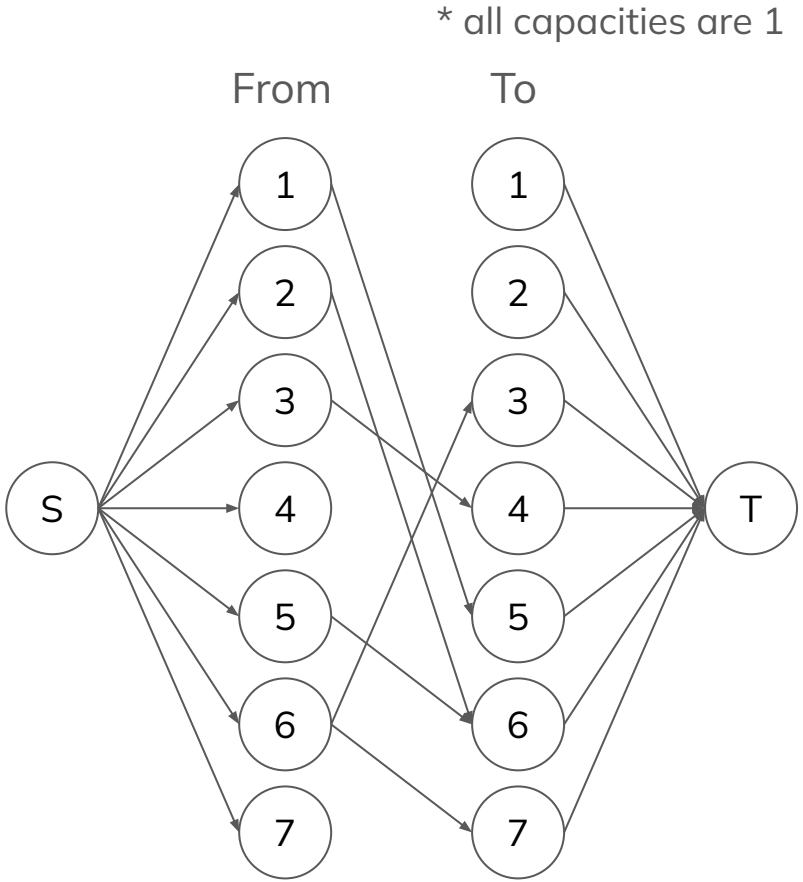




Modeling Problems with Matching

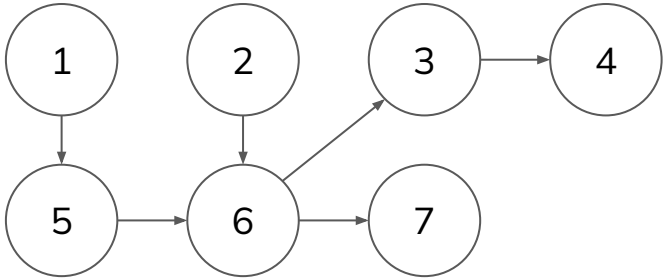
- Create a “from” vertex (out-edge) and “to” vertex (in-edge) for each vertex.
- For each edge u → v, build an edge `u.from` → `v.to`.
- Answer = `|V|` - maximum matching.

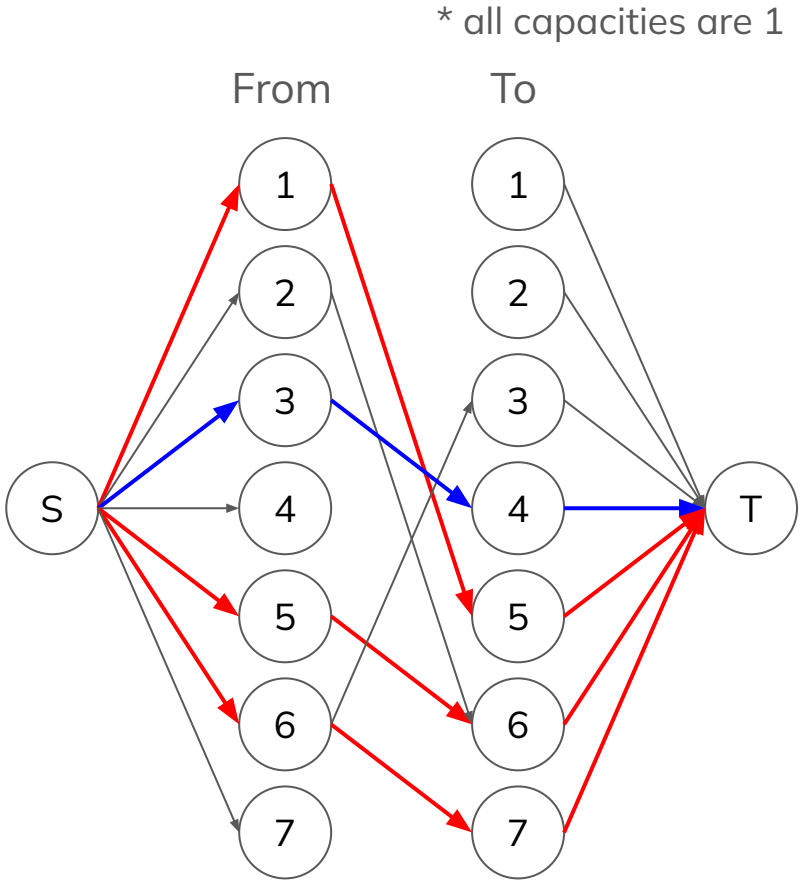




Modeling Problems with Matching

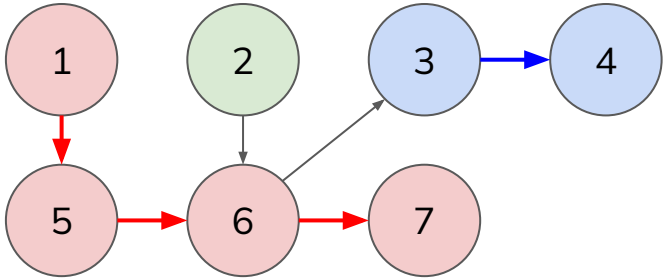
- Create a “from” vertex (out-edge) and “to” vertex (in-edge) for each vertex.
- For each edge u → v, build an edge `u.from` → `v.to`.
- Answer = `|V|` - maximum matching.

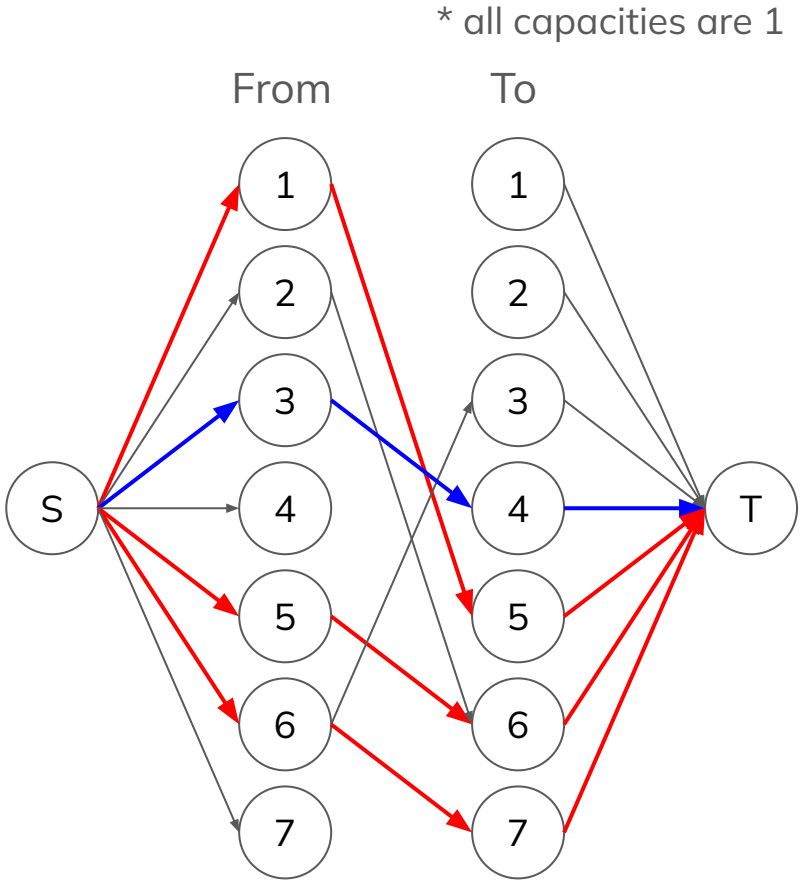




Modeling Problems with Matching

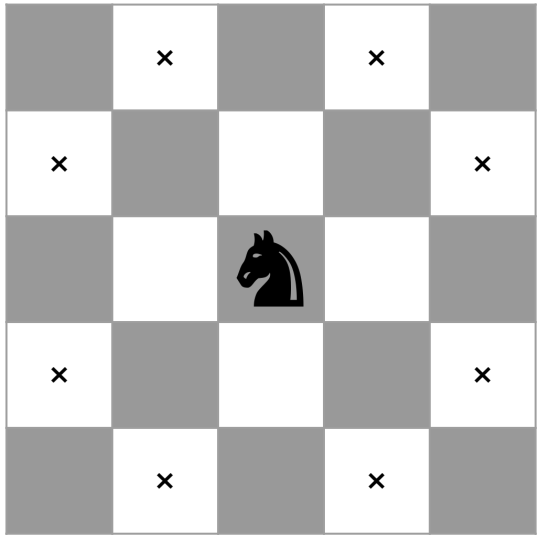
- Create a “from” vertex (out-edge) and “to” vertex (in-edge) for each vertex.
- For each edge u → v, build an edge `u.from` → `v.to`.
- Answer = `|V|` - maximum matching.



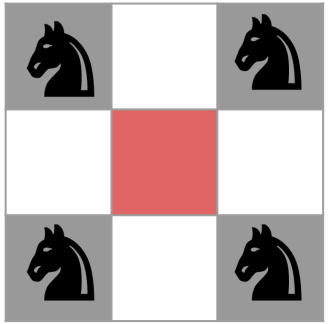


Modeling Problems with Matching

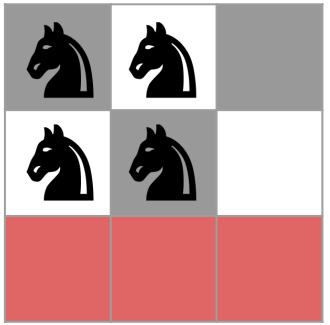
Knights Covering Problem: Given a chessboard with some blocked cells, find the maximum number of knights residing on the chessboard such that none of the knights threaten another knight.



Examples:



Answer = 4



Answer = 4

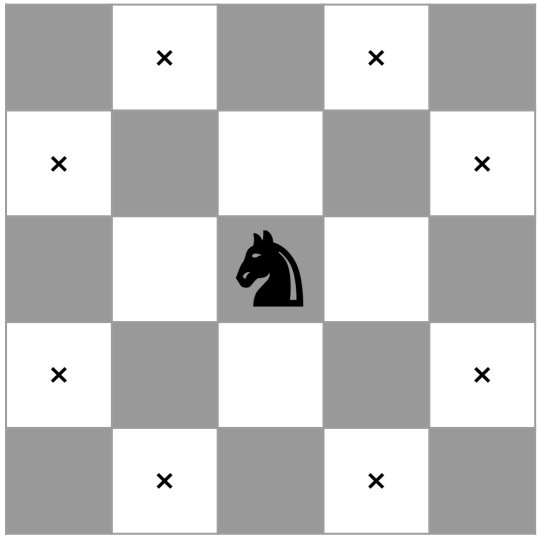
Quick Check

🔠 Which of the following problem is **closest** to the Knights Covering Problem?

A. Maximum Bipartite Matching
B. Minimum Cut
C. Minimum Vertex Cover
D. Maximum Independent Set

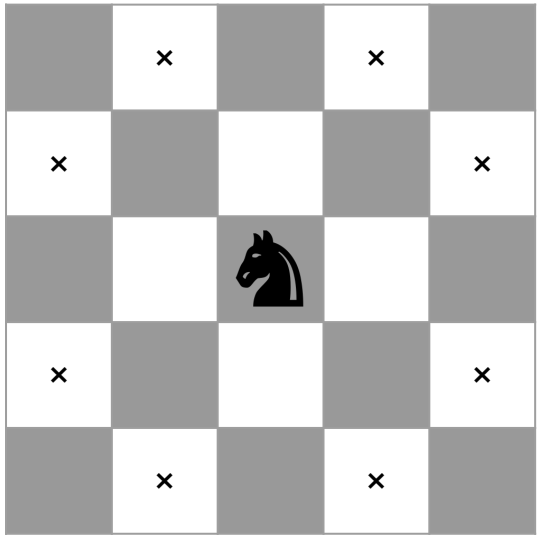
Modeling Problems with Matching

- This is the **maximum independent set** problem on a grid (chessboard).
- Why is the graph bipartite? (How can we find the two set?)



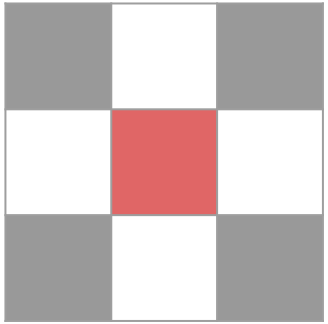
Modeling Problems with Matching

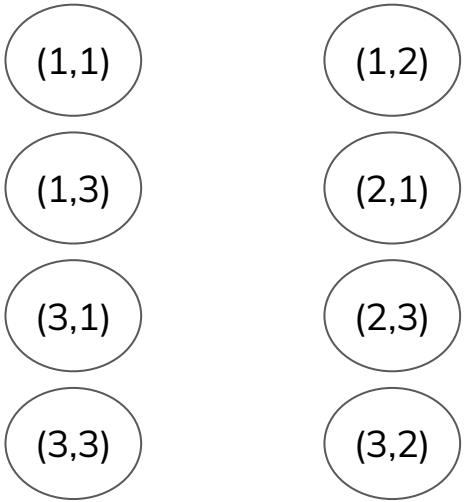
- This is the **maximum independent set** problem on a grid (chessboard).
- Why is the graph bipartite? (How can we find the two set?) Black and white cells
- A knight only threatens knights on cells of the opposite color.



Modeling Problems with Matching

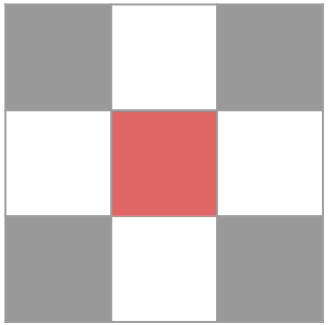
- This is the **maximum independent set** problem on a grid (chessboard).
- Why is the graph bipartite? (How can we find the two set?) Black and white cells
- A knight only threatens knights on cells of the opposite color.

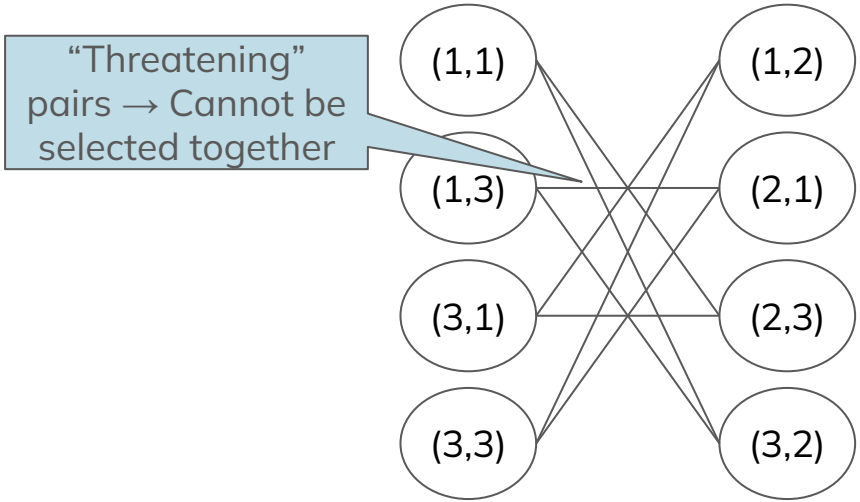




Modeling Problems with Matching

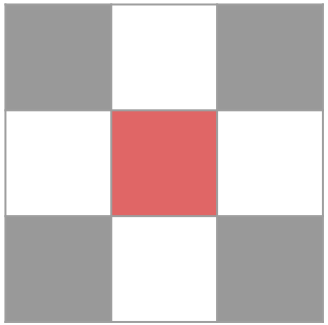
- This is the **maximum independent set** problem on a grid (chessboard).
- Why is the graph bipartite? (How can we find the two set?) Black and white cells
- A knight only threatens knights on cells of the opposite color.

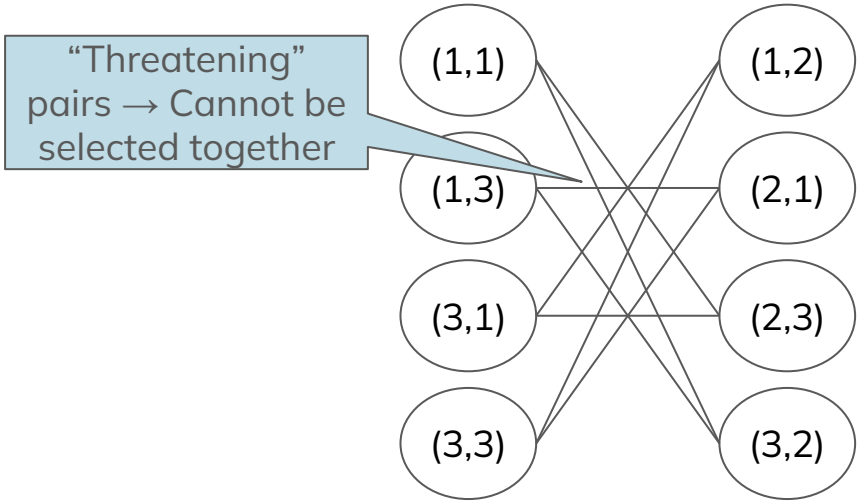




Modeling Problems with Matching

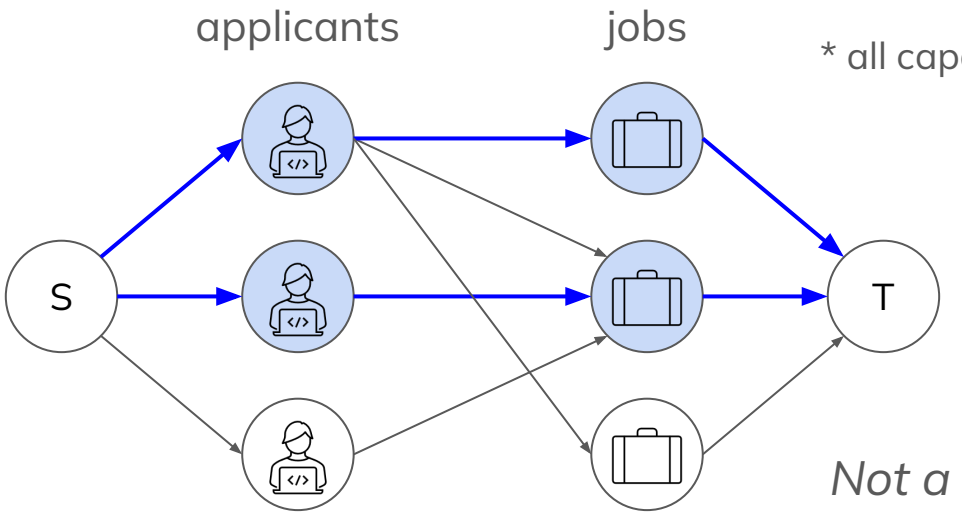
- This is the **maximum independent set** problem on a grid (chessboard).
- Why is the graph bipartite? (How can we find the two set?) Black and white cells
- A knight only threatens knights on cells of the opposite color.
- Answer = Max. Independent Set → Flip(Min. Vertex Cover) → Min. Cut → Max. Flow.





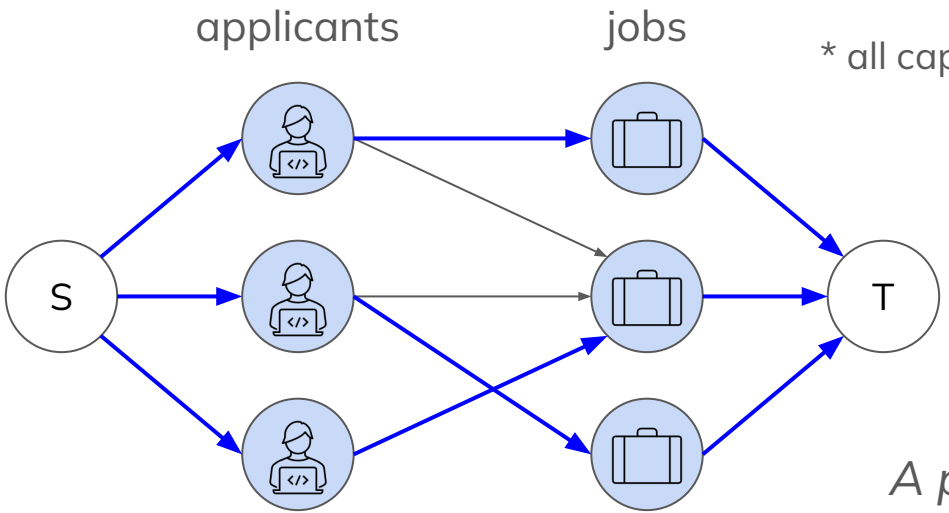
Perfect Matching

We call a matching a **perfect matching** if **all** applicants **or** jobs are matched.



Perfect Matching

We call a matching a **perfect matching** if **all** applicants **or** jobs are matched.

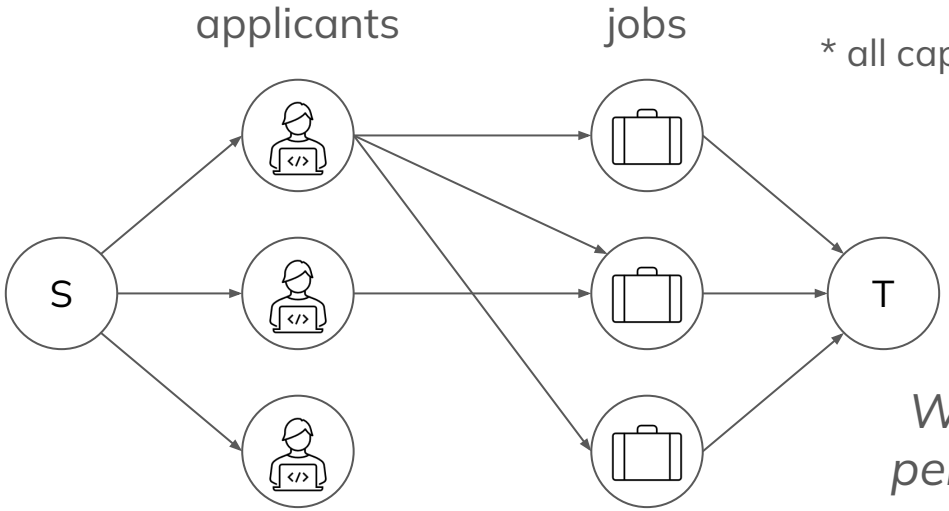


A perfect matching

Hall's Marriage Theorem – Intuition

How can we determine if there exists a perfect matching?

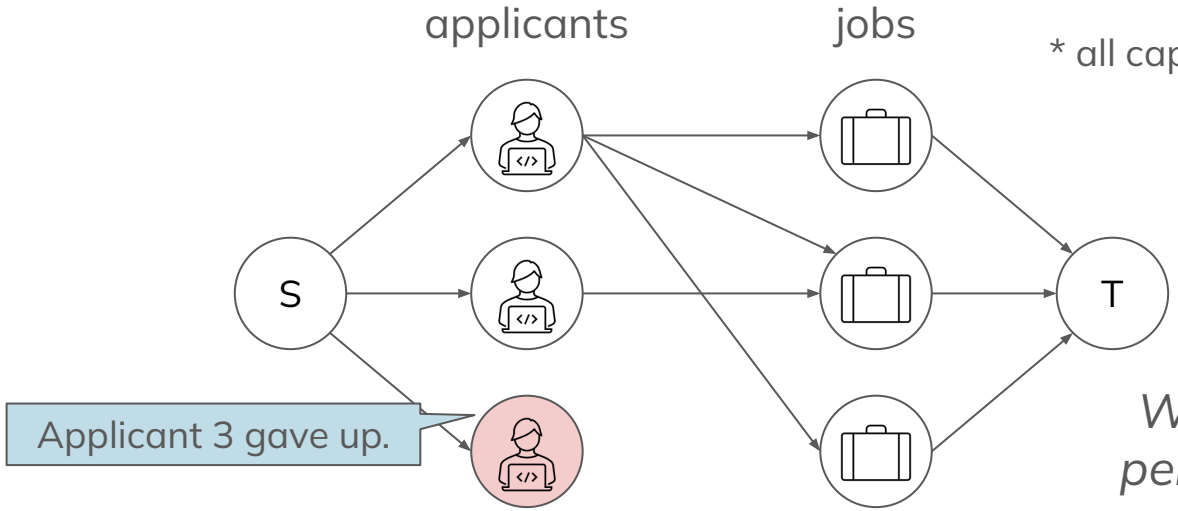
- Yes, you can solve this with maximum flow. // O(EF) = O(NM × min(N, M))
- What if one of the sides is extremely small?



Hall's Marriage Theorem – Intuition

How can we determine if there exists a perfect matching?

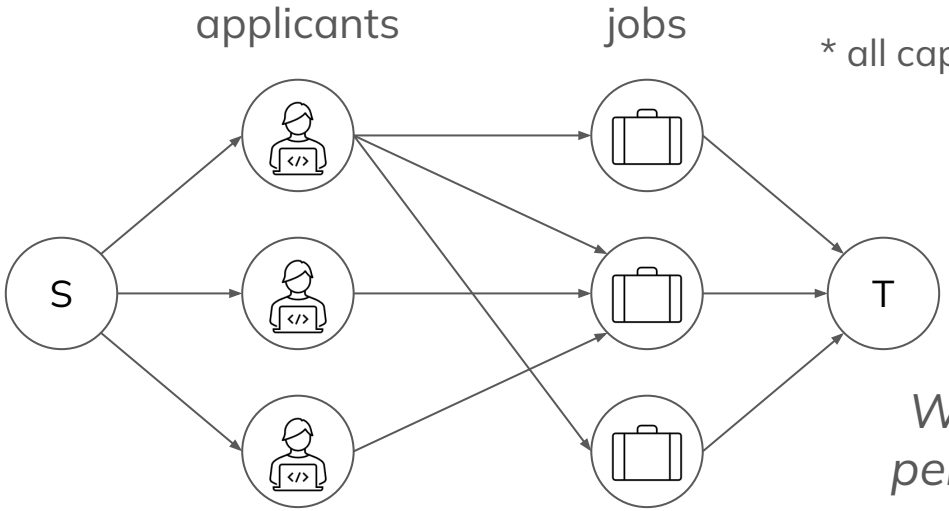
- Yes, you can solve this with maximum flow. // O(EF) = O(NM × min(N, M))
- What if one of the sides is extremely small?



Hall's Marriage Theorem – Intuition

How can we determine if there exists a perfect matching?

- Yes, you can solve this with maximum flow. // O(EF) = O(NM × min(N, M))
- What if one of the sides is extremely small?



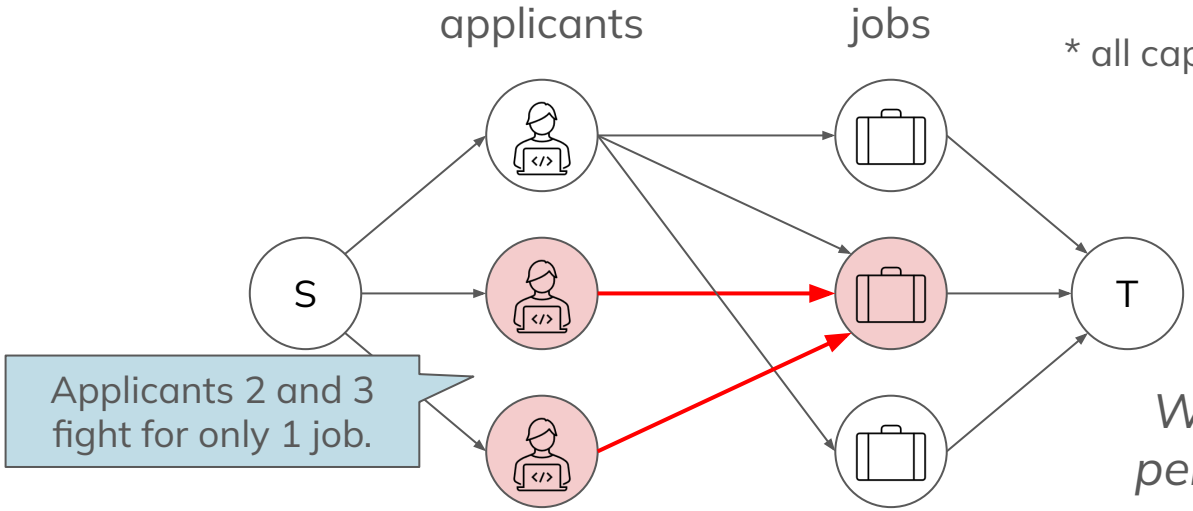
* all capacities are 1

Why are there no perfect matchings?

Hall's Marriage Theorem – Intuition

How can we determine if there exists a perfect matching?

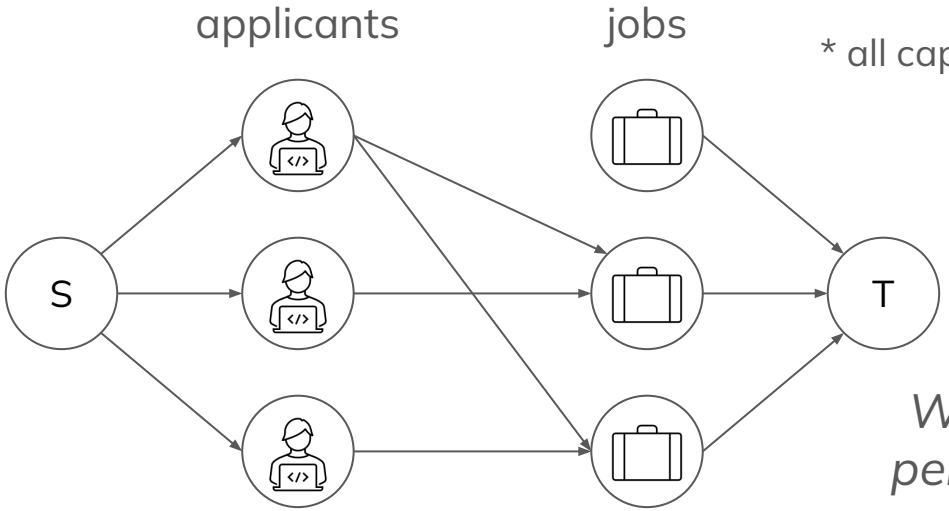
- Yes, you can solve this with maximum flow. // O(EF) = O(NM × min(N, M))
- What if one of the sides is extremely small?



Hall's Marriage Theorem – Intuition

How can we determine if there exists a perfect matching?

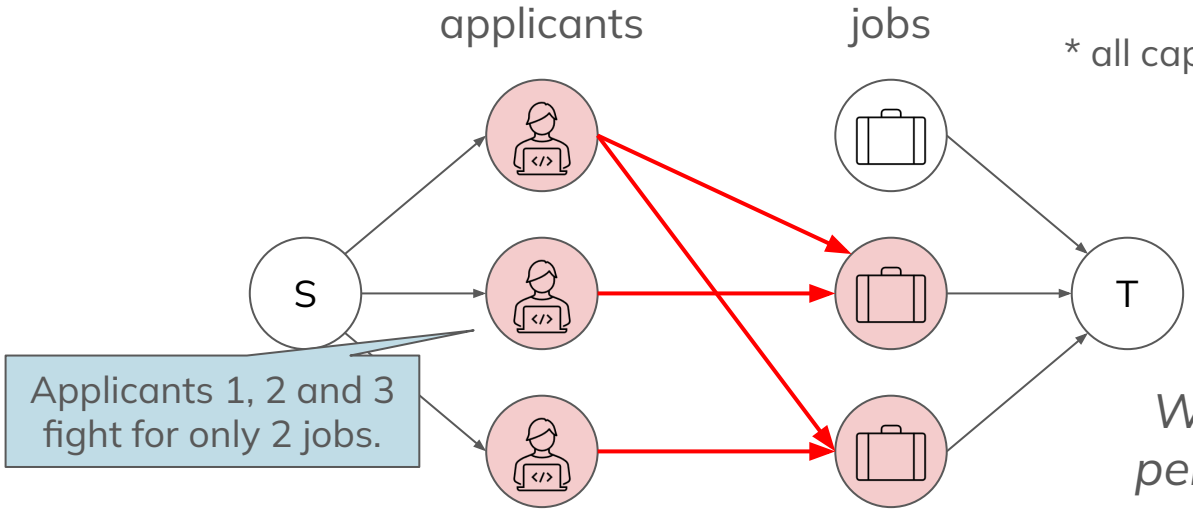
- Yes, you can solve this with maximum flow. // O(EF) = O(NM × min(N, M))
- What if one of the sides is extremely small?



Hall's Marriage Theorem – Intuition

How can we determine if there exists a perfect matching?

- Yes, you can solve this with maximum flow. // O(EF) = O(NM × min(N, M))
- What if one of the sides is extremely small?



Hall's Marriage Theorem

Hall's Marriage Theorem

A bipartite graph has a **perfect matching** **if and only if**:

> For every **subset** of applicants of size k, the applicants apply for at least k **unique** jobs.

- Checking for this condition directly leads us to an $O(2^N \times NM)$ algorithm.
- However, this could be optimized in specific scenarios (e.g. with edge updates).

Summary

To solve **matching** problems:

- Find out the **problem** you are trying to solve (maximum matching / minimum vertex cover / maximum independent set).
- Justify why the graph is **bipartite**.
- Code the maximum matching algorithm and transform the solution using:

  Max. Matching ↔ Min. Cut ↔ Min. Vertex Cover ↔ Max. Independent Set

Flows with Demands

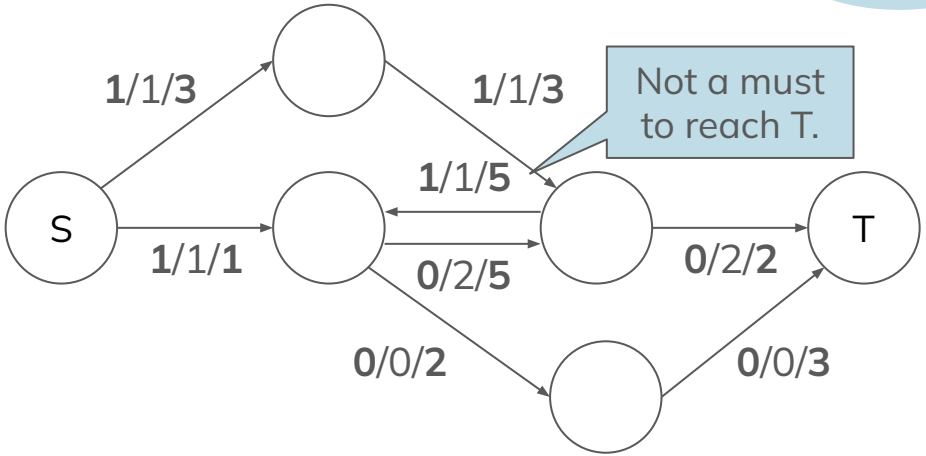
Flows with Demands

You are given a flow network, where each edge has a **demand** and a **capacity**.

- **Demand**: Minimum amount of flow via the edge.
- **Capacity**: Maximum amount of flow via the edge.

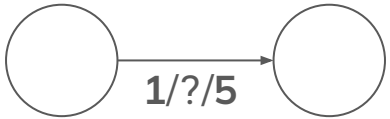
Find **any** flow that satisfies all constraints.

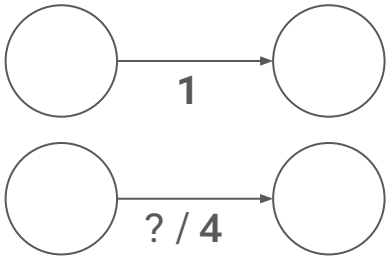
Note that here, "constraints" also mean **total flow in = total flow out** for every node.



Flows with Demands – Solution

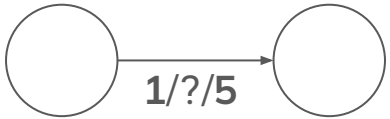
- Intuition: Each edge can be broken down into two edges, **compulsory** flow and **optional** flow.

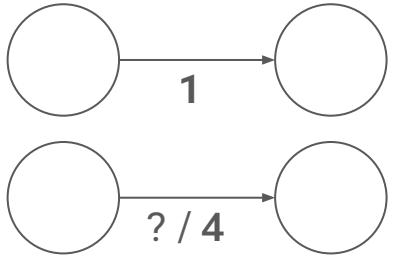




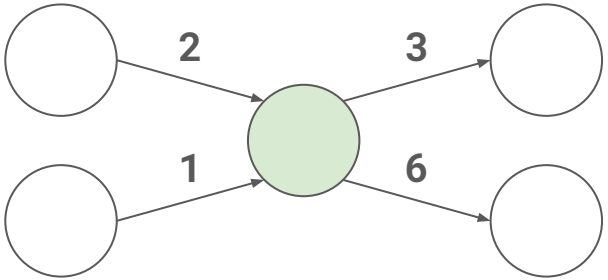
Flows with Demands – Solution

- Intuition: Each edge can be broken down into two edges, **compulsory** flow and **optional** flow.



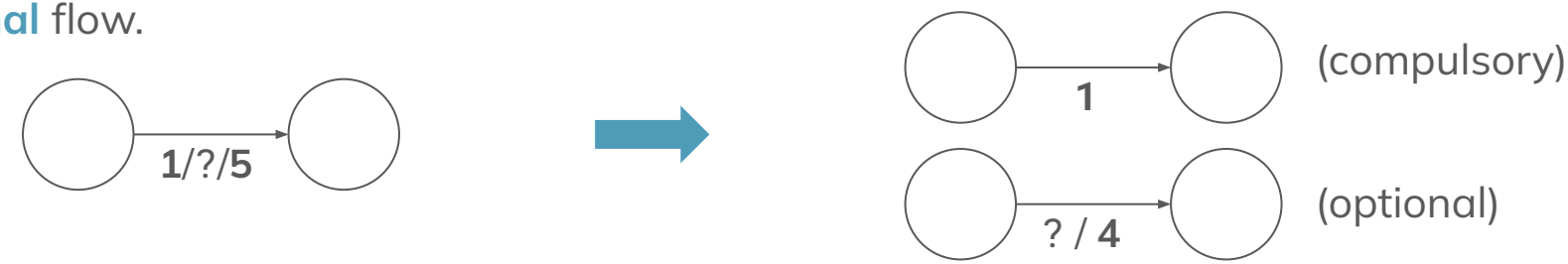


- For each vertex (other than S and T), there is compulsory flow in/out:



Flows with Demands – Solution

- Intuition: Each edge can be broken down into two edges, **compulsory** flow and **optional** flow.

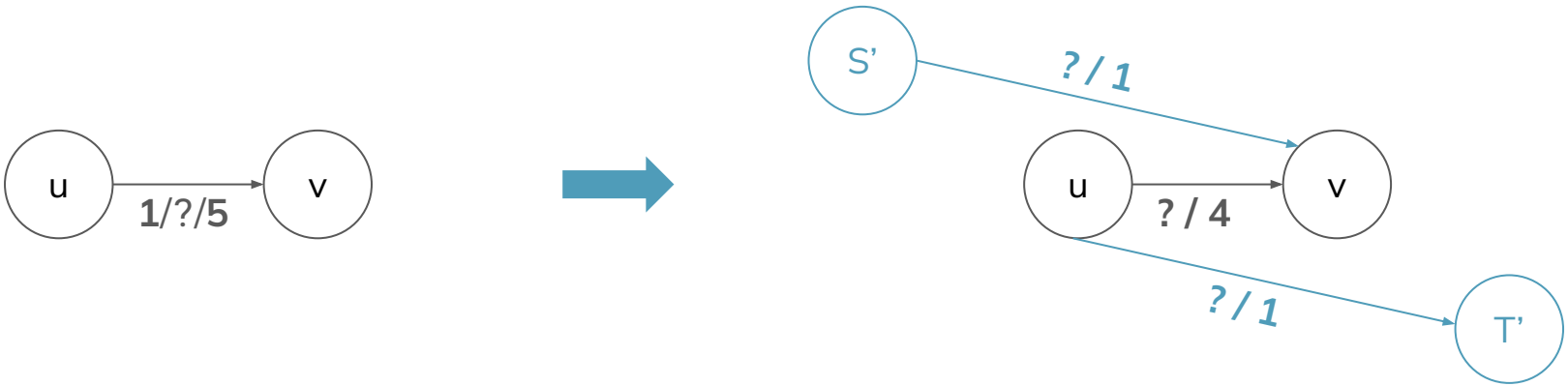


- For each vertex (other than S and T), there is compulsory flow in/out:
  - Send this "flow in" and "flow out" directly from a **super source/sink**.

(control saturation directly with "maximum flow")

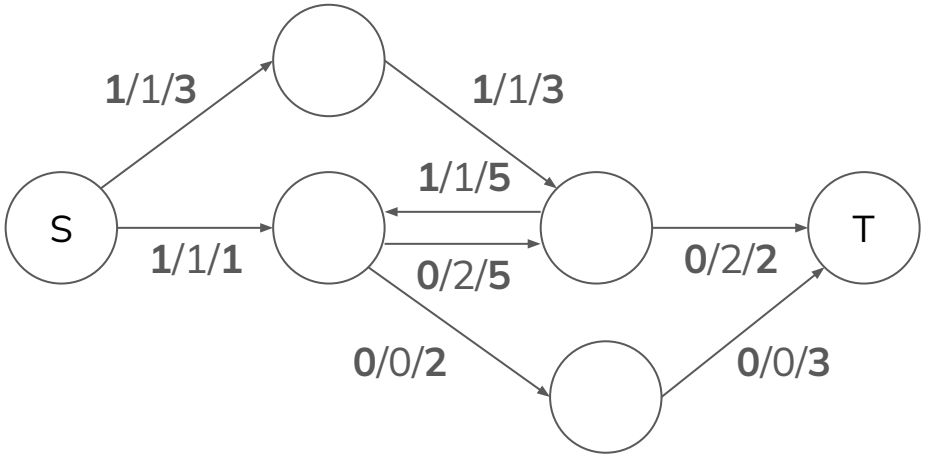
Flows with Demands – Solution

- For each edge u → v, build the "optional" edge (**Capacity - Demand**)… and build the "compulsory" edges S' → v and u → T' (**Demand** only).



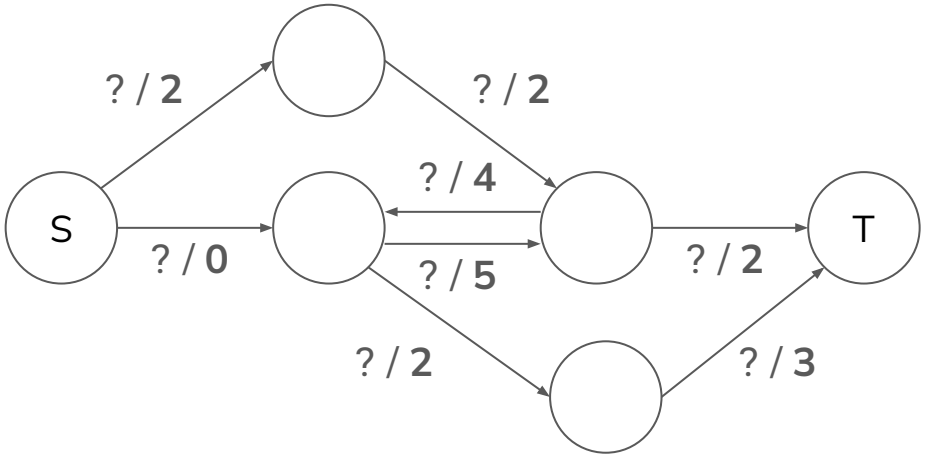
Flows with Demands – Solution

- For each edge u → v, build the "optional" edge (**Capacity - Demand**)... and build the "compulsory" edges S' → v and u → T' (**Demand** only).



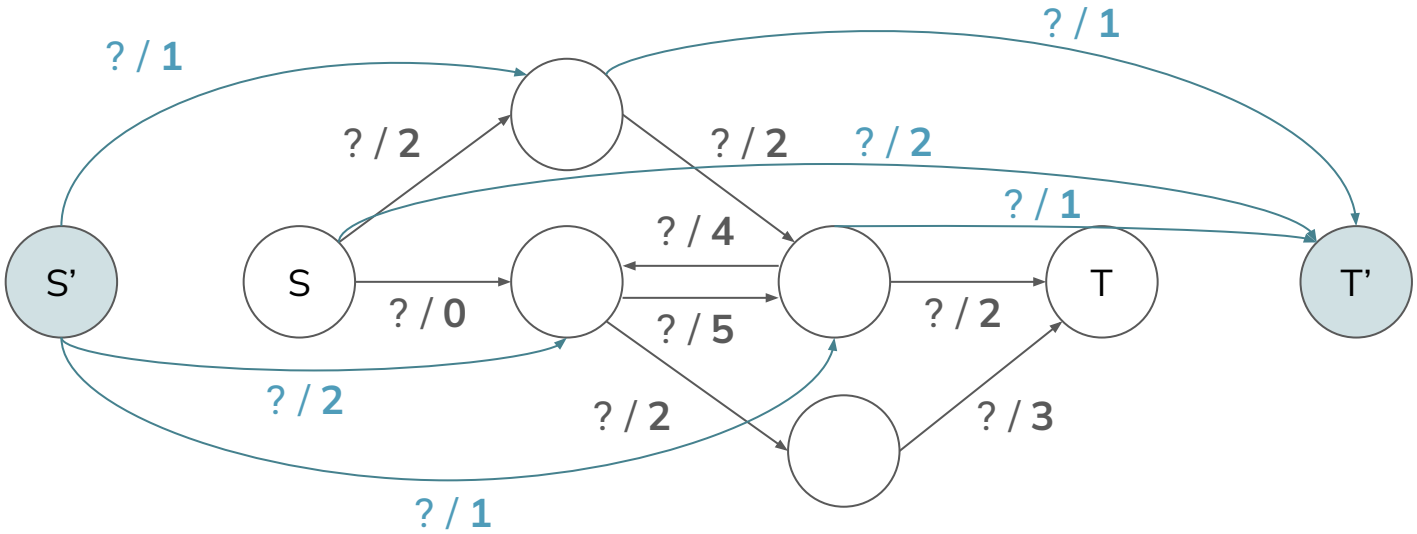
Flows with Demands – Solution

- For each edge u → v, build the “optional” edge (**Capacity - Demand**)... and build the “compulsory” edges S’ → v and u → T’ (**Demand** only).



Flows with Demands – Solution

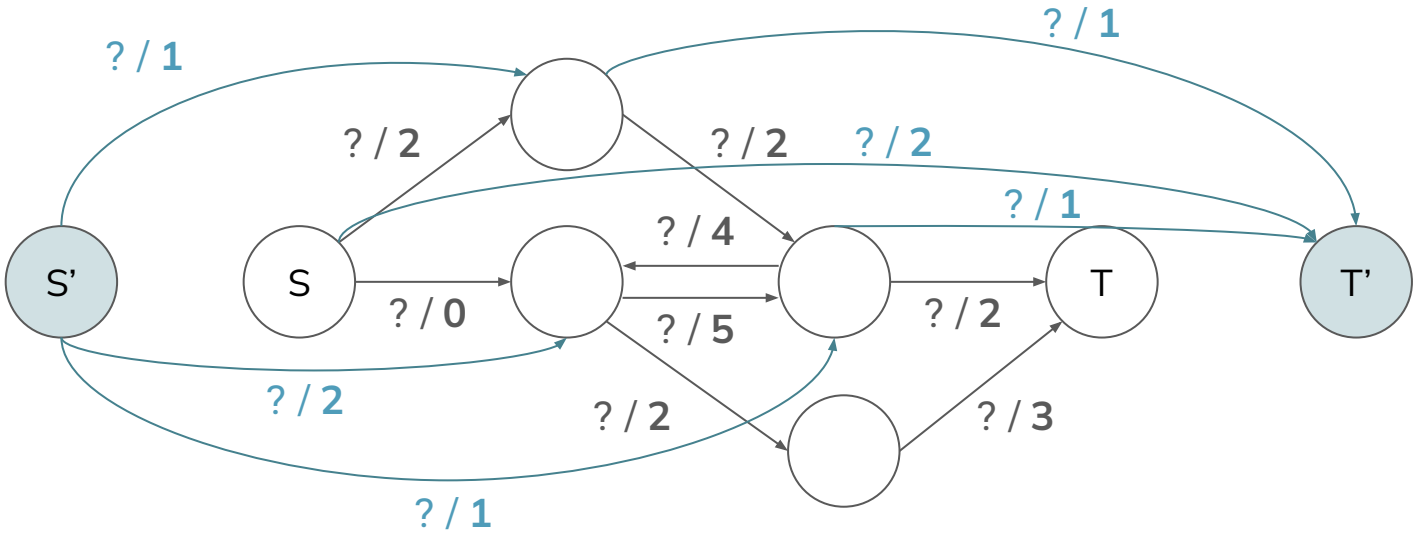
- For each edge u → v, build the "optional" edge (**Capacity - Demand**)... and build the "compulsory" edges S' → v and u → T' (**Demand** only).



Flows with Demands – Solution

- For each edge u → v, build the "optional" edge (**Capacity - Demand**)...
  and build the "compulsory" edges S' → v and u → T' (**Demand** only).

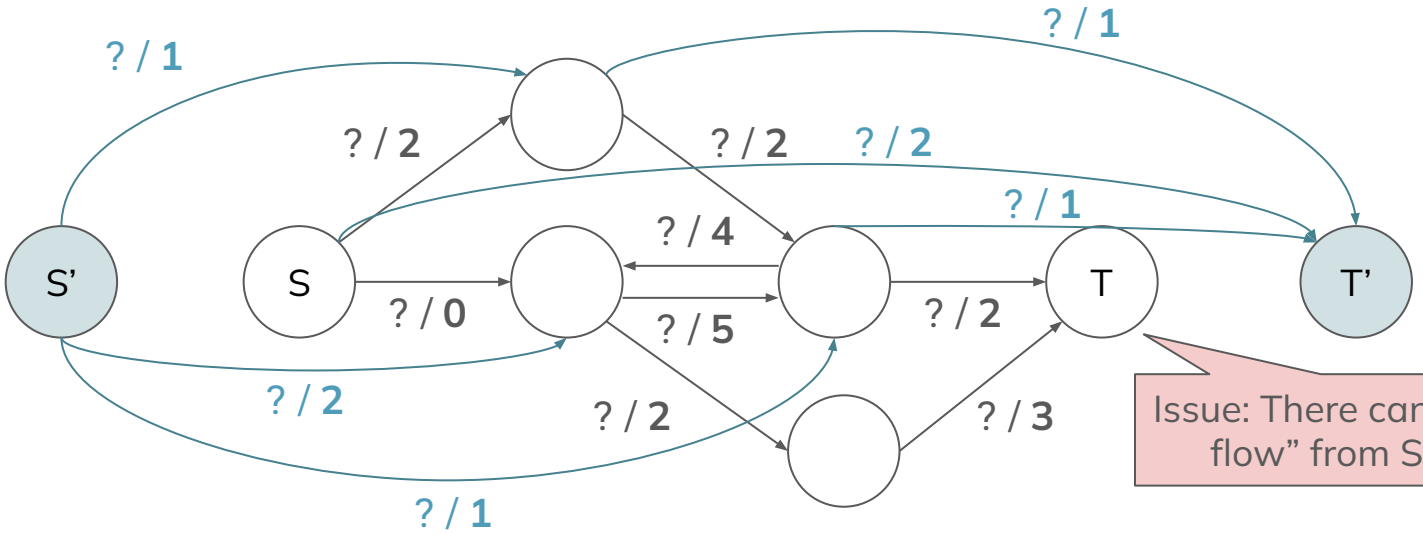
Maximum flow from S' to T' should be 4 (sum of demands).



Flows with Demands – Solution

- For each edge u → v, build the "optional" edge (**Capacity - Demand**)…
  and build the "compulsory" edges S' → v and u → T' (**Demand** only).

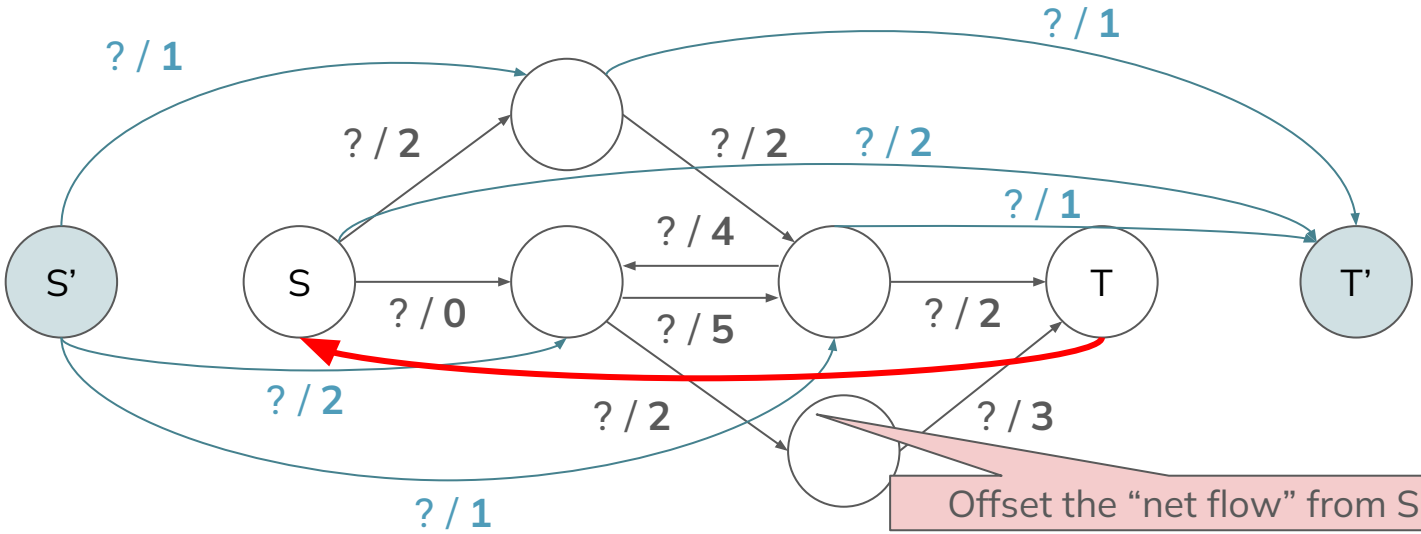
Maximum flow from S' to T' should be 4 (sum of demands).



Flows with Demands – Solution

- For each edge u → v, build the "optional" edge (**Capacity - Demand**)...
  and build the "compulsory" edges S' → v and u → T' (**Demand** only).

Maximum flow from S' to T' should be 4 (sum of demands).

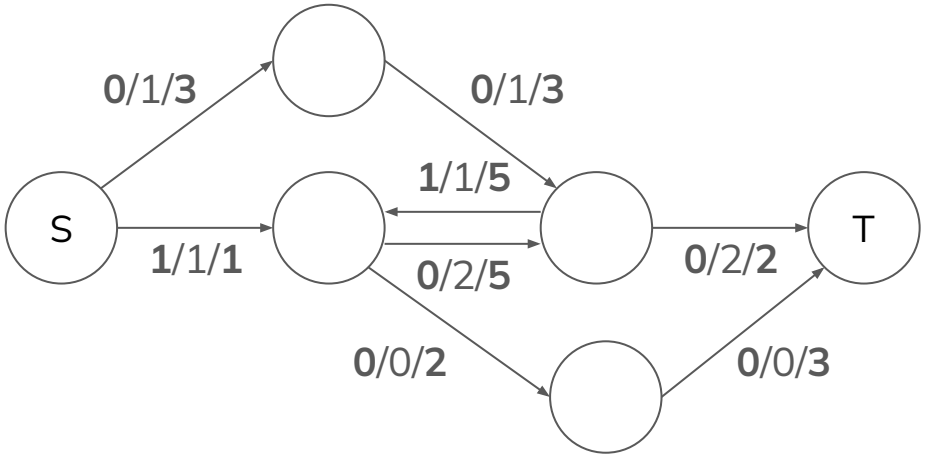


Flows with Demands – Minimum Flow

You are given a flow network, where each edge has a **demand** and a **capacity**.

- **Demand**: Minimum amount of flow via the edge.
- **Capacity**: Maximum amount of flow via the edge.

Find some flow that satisfies all constraints and **minimizes the flow from S to T**.



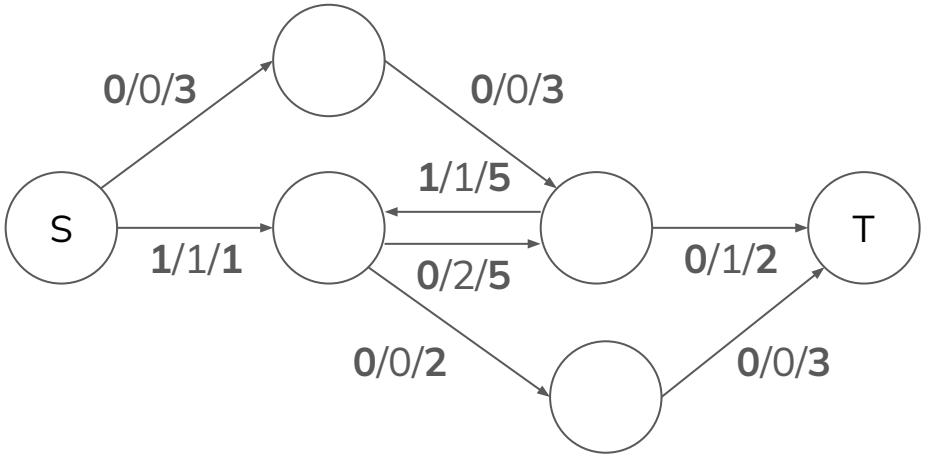
Valid but not minimized

Flows with Demands – Minimum Flow

You are given a flow network, where each edge has a **demand** and a **capacity**.

- **Demand**: Minimum amount of flow via the edge.
- **Capacity**: Maximum amount of flow via the edge.

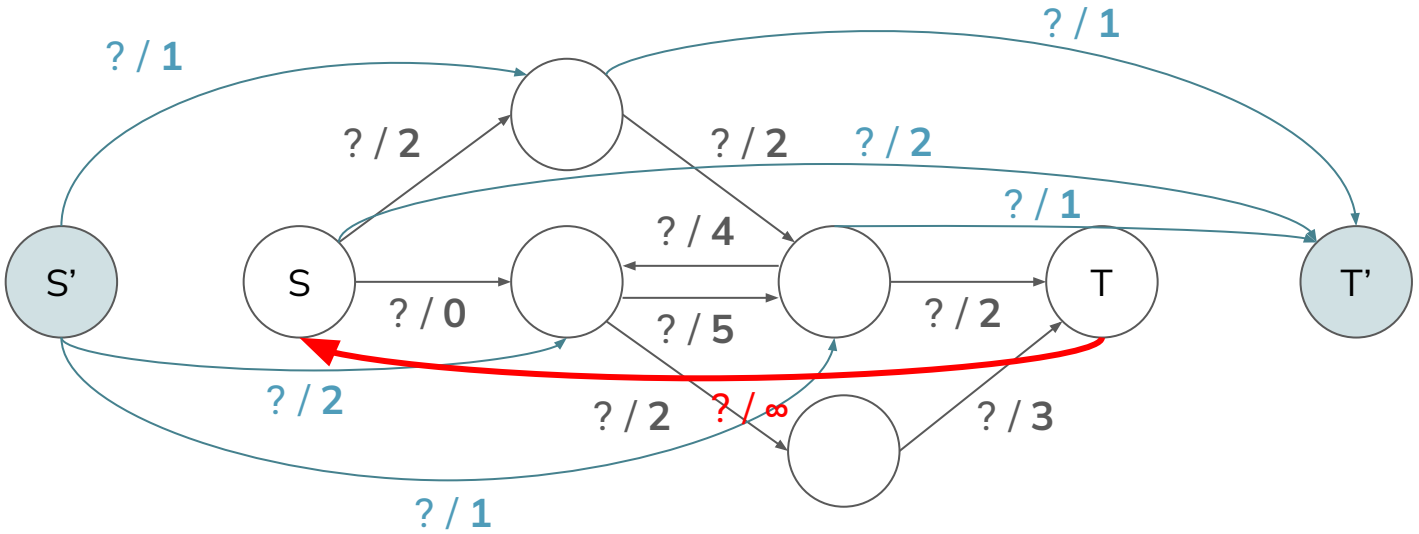
Find some flow that satisfies all constraints and **minimizes the flow from S to T**.



Valid and minimized

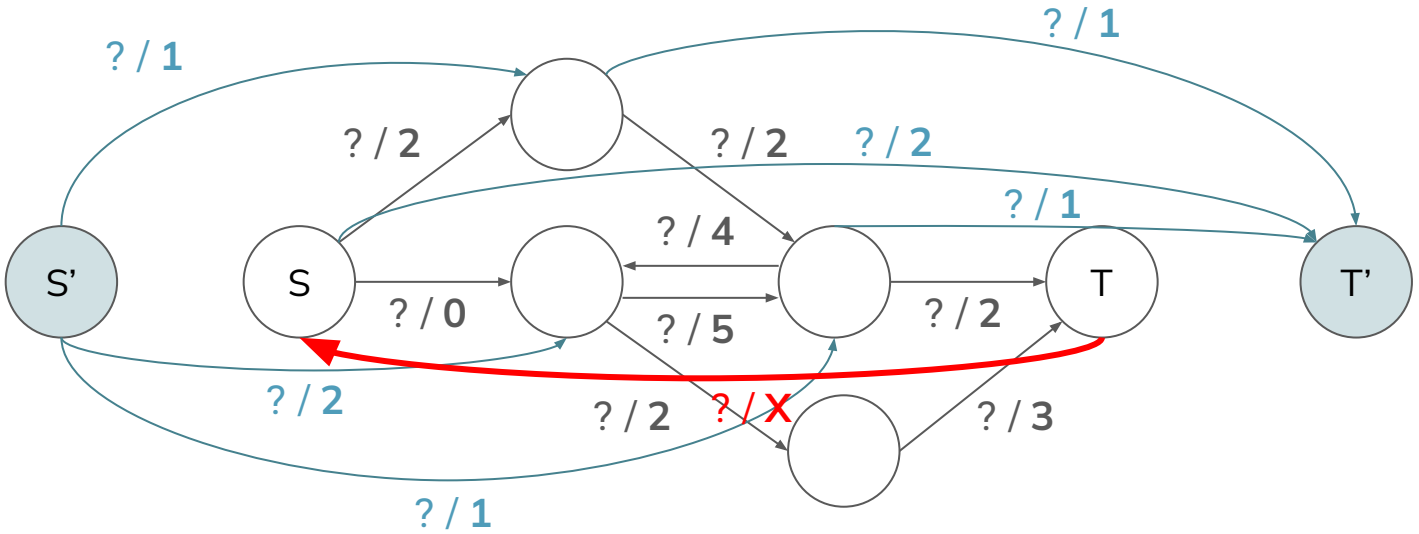
Flows with Demands – Minimum Flow

- Remember that the edge T → S offsets the flow from S to T.



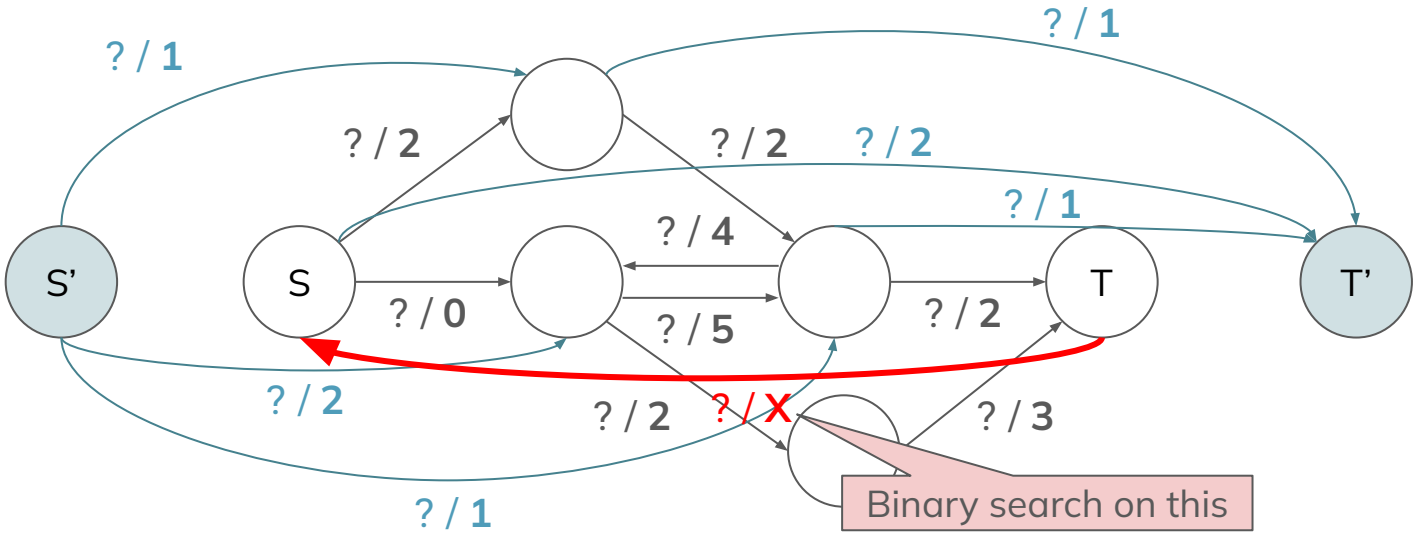
Flows with Demands – Minimum Flow

- Remember that the edge T → S offsets the flow from S to T.
- By varying the capacity of this edge, we can control the maximum flow from S to T.



Flows with Demands – Minimum Flow

- Remember that the edge T → S offsets the flow from S to T.
- By varying the capacity of this edge, we can control the maximum flow from S to T.
  ⇒ **Binary search on answer**.



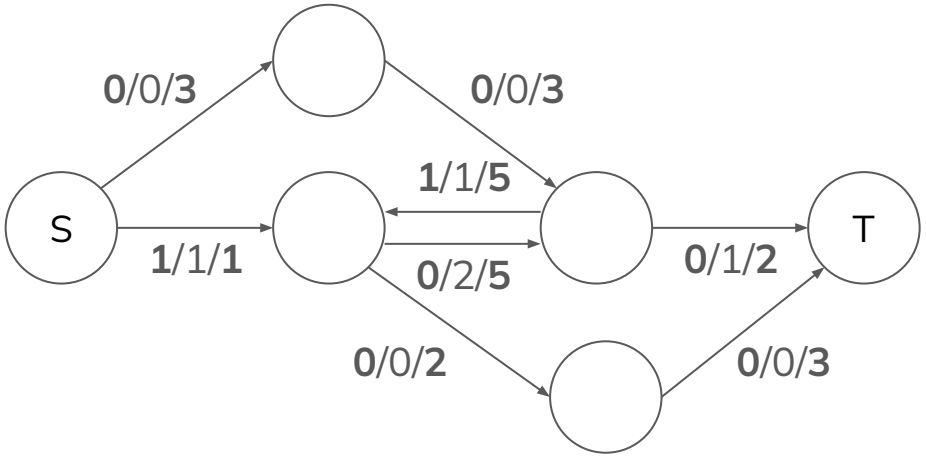
Flows with Demands – Maximum Flow

You are given a flow network, where each edge has a **demand** and a **capacity**.

- **Demand**: Minimum amount of flow via the edge.
- **Capacity**: Maximum amount of flow via the edge.

Find some flow that satisfies all constraints and **maximizes the flow from S to T**.

Completed: Arbitrary Flow



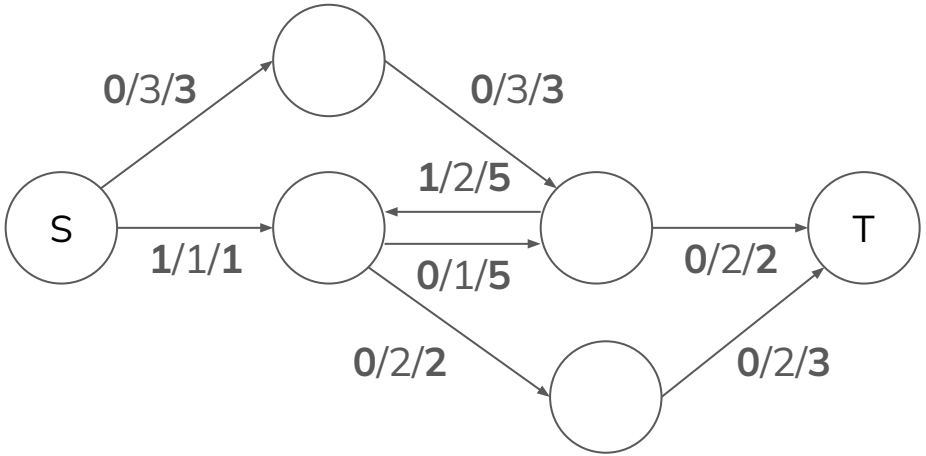
Once we've found an arbitrary flow... Continue finding augmenting paths on the **original flow network**!

Flows with Demands – Maximum Flow

You are given a flow network, where each edge has a **demand** and a **capacity**.

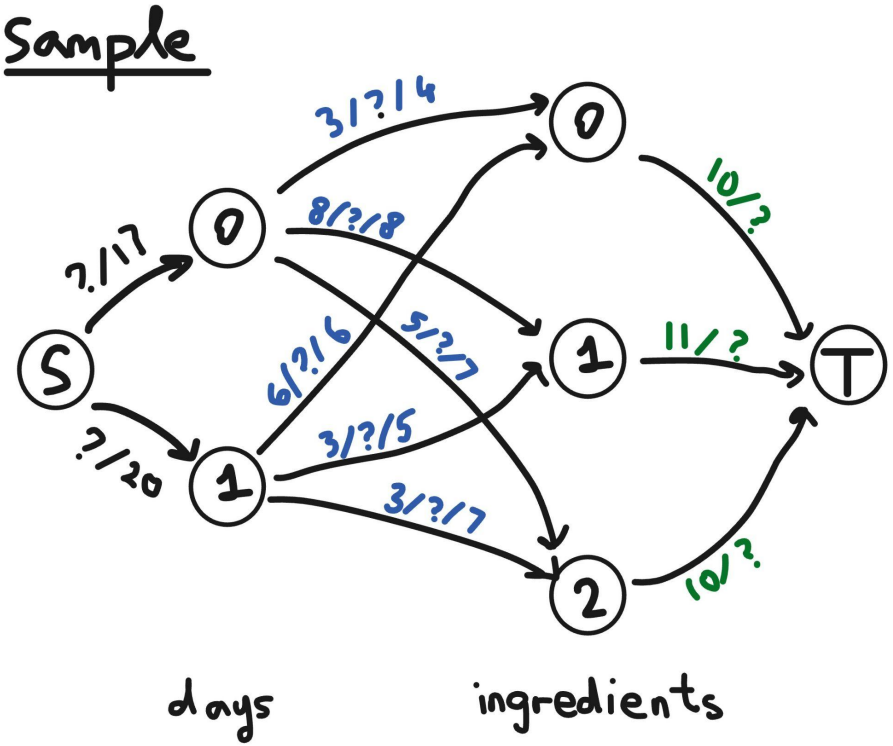
- **Demand**: Minimum amount of flow via the edge.
- **Capacity**: Maximum amount of flow via the edge.

Find some flow that satisfies all constraints and **maximizes the flow from S to T**.



Maximum Flow

Hint to Example Task: A422 Flows with Demands



Task List

Bipartite Matching

- **A420 Maximum Bipartite Matching**
- P3355 騎士共存問題
- M24BF From Programming With Love

Min. Vertex Cover & Max. Independent Set

- **A421 Min. Vertex Cover & Max. Independent Set**
- 01703 Asteroids
- CF2026E - Best Subsequence

Hall's Marriage Theorem

- CF628F - Bear and Fair Set
- CF1009G - Allowed Letters
- CF981F - Round Marriage

Flows with Demands

- **A422 Flows with Demands**
- ABC285G - Tatami

More Tasks

- 洛谷網絡流與線性規劃 24題

Acknowledgements & Extra Resources

Acknowledgements:

- Benson Yeung {mtyeung1} for making the slides

Extra Resources:

- Simulated Cost Flow (2023 Training Camp) by Ethen Yuen {ethening}
- CP-Algorithms article on Flows with Demands
- P3386【模板】二分圖最大匹配 題解 ← Pretty intuitive explanation on how to solve the Bipartite Matching Problem
- Kőnig's and Hall's theorems through minimum cut in bipartite graphs by adamant

Team Problem Set

Team Problem Set

- Try to work on graph modeling!
  - Teams of 3, work collaboratively!
  - Duration: Around 1 hour (depends on lesson time)

- Work on the same question *together*, as opposed to divide & conquer strategies.
  - Time given should be more than enough.
  - Team members should understand each other's solutions.

- Draw the flow network + Describe how to obtain the answer in 1 sentence.

- Teams will be invited to present their solutions to the whole class.