Maximum Bipartite Matching

Robin Visser

IOI Training Camp
University of Cape Town

4 February 2017

Overview

Definition

A **bipartite graph** is a graph where each node belongs to one of two disjoint sets, U or V, and every edge connects a node in U to a node in V.

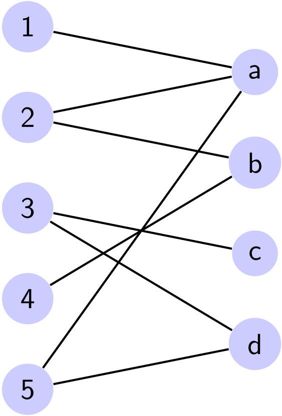
Definition

A **bipartite graph** is a graph where each node belongs to one of two disjoint sets, U or V, and every edge connects a node in U to a node in V.

A **maximal bipartite matching** is the largest subset of edges in a bipartite graph such that no two selected edges share a common vertex.

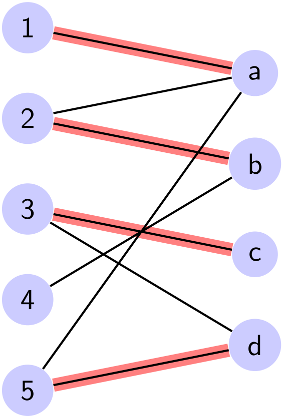
Bipartite Graph Example

Given example with 5 left nodes and 4 right nodes:



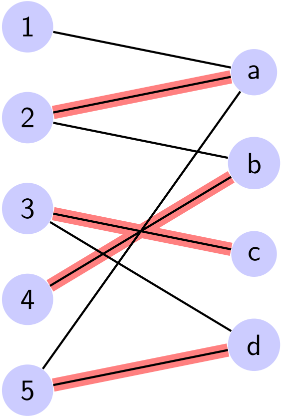
Maximal Matching Example

Given example with 5 left nodes and 4 right nodes:



Maximal Matching Example

Matching not guaranteed to be unique.



Network Flow Approach

We can solve the maximum bipartite matching problem using a *network flow* approach.

Network Flow Approach

We can solve the maximum bipartite matching problem using a *network flow* approach.

- We first ensure that all edges from U to V are directed.

Network Flow Approach

We can solve the maximum bipartite matching problem using a *network flow* approach.

- We first ensure that all edges from U to V are directed.
- We introduce a single **source** node that has outgoing edges to each node in U

Network Flow Approach

We can solve the maximum bipartite matching problem using a *network flow* approach.

- We first ensure that all edges from U to V are directed.
- We introduce a single **source** node that has outgoing edges to each node in U
- We then add a **sink** node that has incoming edges from each node in V.

Network Flow Approach

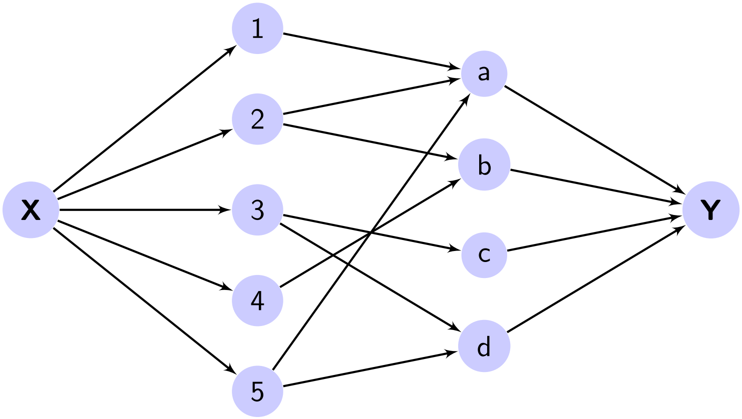
We can solve the maximum bipartite matching problem using a *network flow* approach.

- We first ensure that all edges from U to V are directed.
- We introduce a single **source** node that has outgoing edges to each node in U
- We then add a **sink** node that has incoming edges from each node in V.

The problem now becomes finding the *maximal flow* in the graph.

Flow Example

We denote the source node as **X** and the sink node as **Y**



Flow Definition

Let's briefly define what the max flow problem is for a general unweighted directed graph:

Flow Definition

Let's briefly define what the max flow problem is for a general unweighted directed graph:

Let (V, E) be a directed graph with a source node $s \in V$ and a sink node $t \in V$. A **flow** is a mapping $f : E \to \{0, 1\}$ which satisfies the following constraint: For each node $v \in V \backslash \{s, t\}$:

$$\sum_{a:(a,v)\in E} f(a, v) = \sum_{b:(v,b)\in E} f(v, b)$$

Flow Definition

Let's briefly define what the max flow problem is for a general unweighted directed graph:

Let (V, E) be a directed graph with a source node $s \in V$ and a sink node $t \in V$. A **flow** is a mapping $f : E \to \{0, 1\}$ which satisfies the following constraint: For each node $v \in V \backslash \{s, t\}$:

$$\sum_{a:(a,v)\in E} f(a, v) = \sum_{b:(v,b)\in E} f(v, b)$$

The value of flow is given as $|f| = \sum_{a:(s,a)\in E} f(s, a)$ (i.e. the total flow coming out of the source node).
The **maximal flow problem** is to simply maximise $|f|$.

Flow Algorithm

One can more intuitively consider flow as water running through pipes represented by the edges.

Flow Algorithm

One can more intuitively consider flow as water running through pipes represented by the edges.

There are many algorithms one could use to solve *network flow* problems (such as Ford–Fulkerson algorithm, Edmonds–Karp algorithm, Dinic's algorithm etc.) For the specific case of **bipartite graphs**, we don't need the full generality of most network flow algorithms.

Flow Algorithm

One can more intuitively consider flow as water running through pipes represented by the edges.

There are many algorithms one could use to solve *network flow* problems (such as Ford–Fulkerson algorithm, Edmonds–Karp algorithm, Dinic's algorithm etc.) For the specific case of **bipartite graphs**, we don't need the full generality of most network flow algorithms.

Since we've converted the problem to an unweighted directed graph, we can therefore use the following simplified algorithm which solves the max flow problem for any general unweighted directed graph.

Flow Algorithm

Solving maximal flow for an unweighted directed graph:

1. Construct the directed graph with a source and sink node as described before.

Flow Algorithm

Solving maximal flow for an unweighted directed graph:

1. Construct the directed graph with a source and sink node as described before.
2. Using a DFS or BFS, find a path from the source to the sink.

Flow Algorithm

Solving maximal flow for an unweighted directed graph:

1. Construct the directed graph with a source and sink node as described before.
2. Using a DFS or BFS, find a path from the source to the sink.
3. Once a path is found, reverse each edge on this path.

Flow Algorithm

Solving maximal flow for an unweighted directed graph:

1. Construct the directed graph with a source and sink node as described before.
2. Using a DFS or BFS, find a path from the source to the sink.
3. Once a path is found, reverse each edge on this path.
4. Repeat step 2 and 3 until no more paths from the sink to source exist.

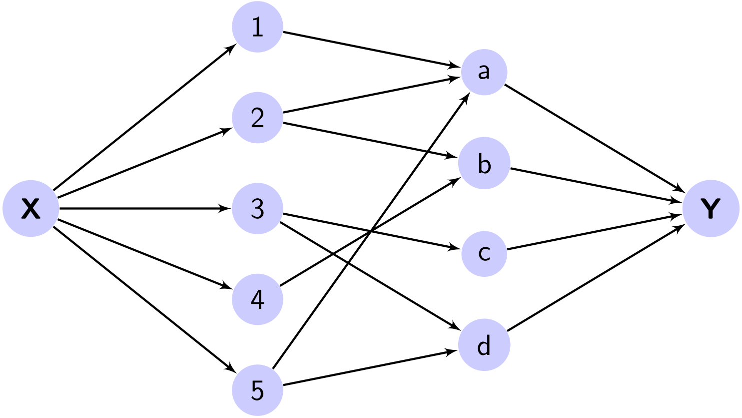
Flow Algorithm

Solving maximal flow for an unweighted directed graph:

1. Construct the directed graph with a source and sink node as described before.
2. Using a DFS or BFS, find a path from the source to the sink.
3. Once a path is found, reverse each edge on this path.
4. Repeat step 2 and 3 until no more paths from the sink to source exist.
5. The final matching solution is then the set of edges between u and V that are reversed (i.e. run from V to U)

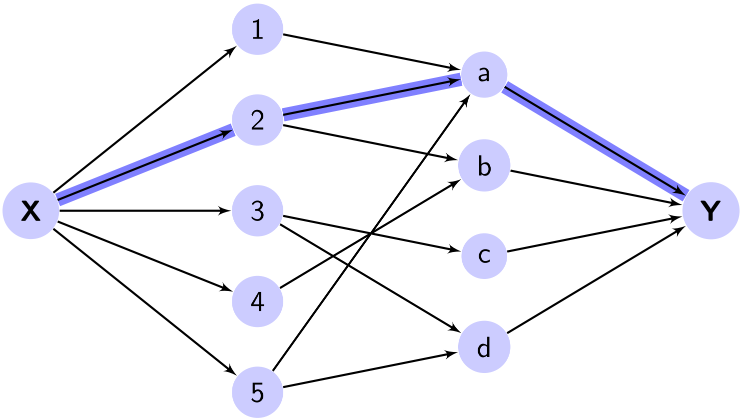
Flow Algorithm

Construct graph



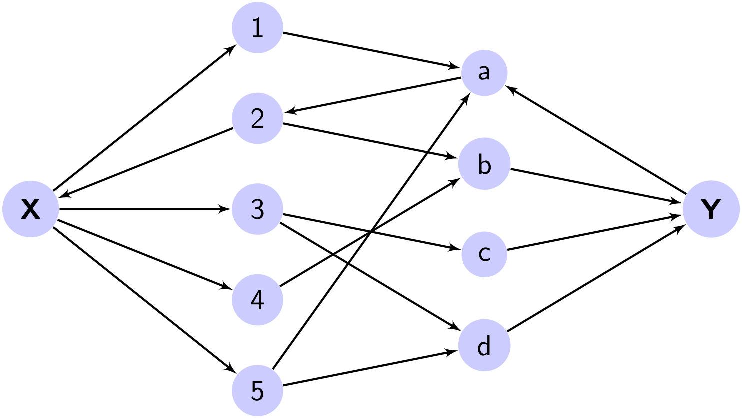
Flow Algorithm

Found path: $\mathbf{X} \to 2 \to \mathrm{a} \to \mathbf{Y}$



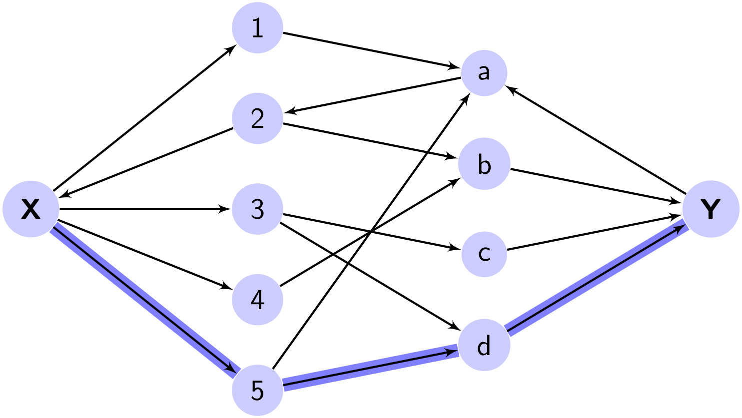
Flow Algorithm

Reverse edges



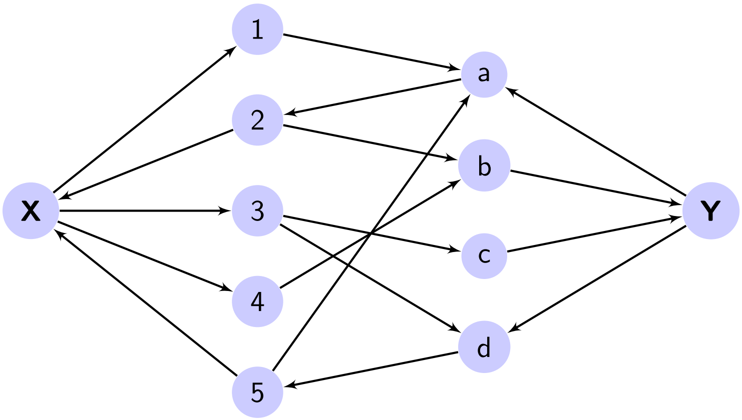
Flow Algorithm

Found path: **X** $\rightarrow$ 5 $\rightarrow$ d $\rightarrow$ **Y**



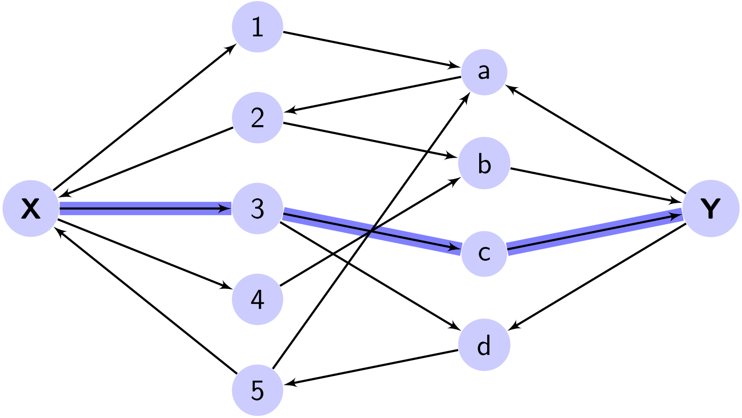
Flow Algorithm

Reverse edges



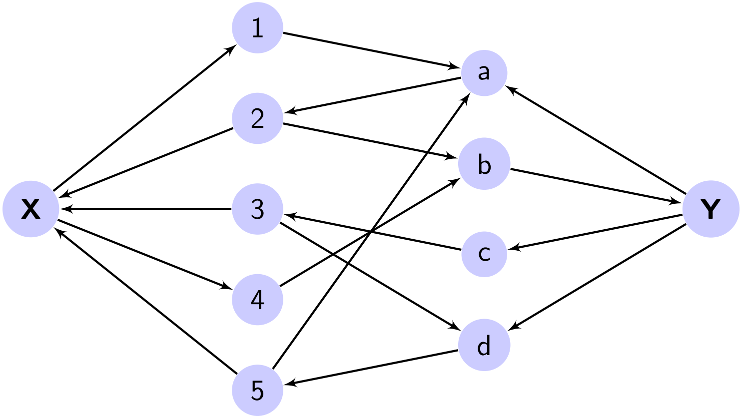
Flow Algorithm

Found path: **X** $\rightarrow$ 3 $\rightarrow$ c $\rightarrow$ **Y**



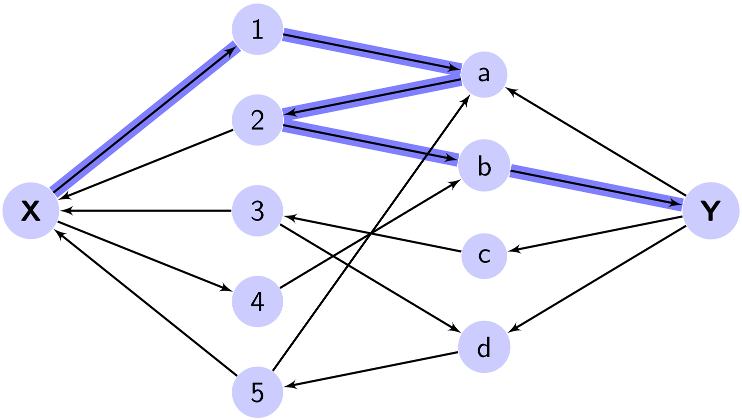
Flow Algorithm

Reverse edges



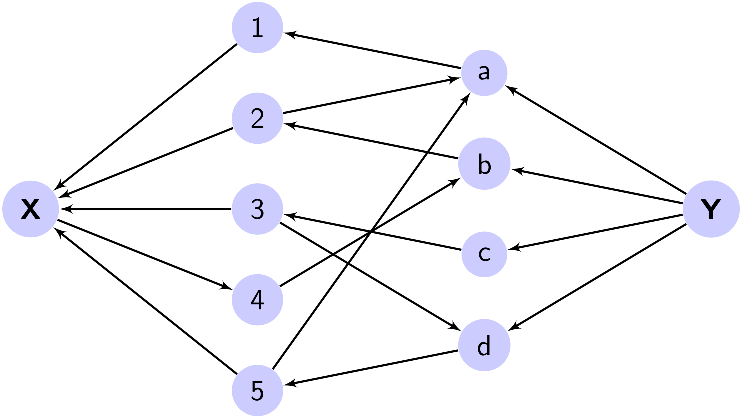
Flow Algorithm

Found path: **X** $\rightarrow$ 1 $\rightarrow$ a $\rightarrow$ 2 $\rightarrow$ b $\rightarrow$ **Y**



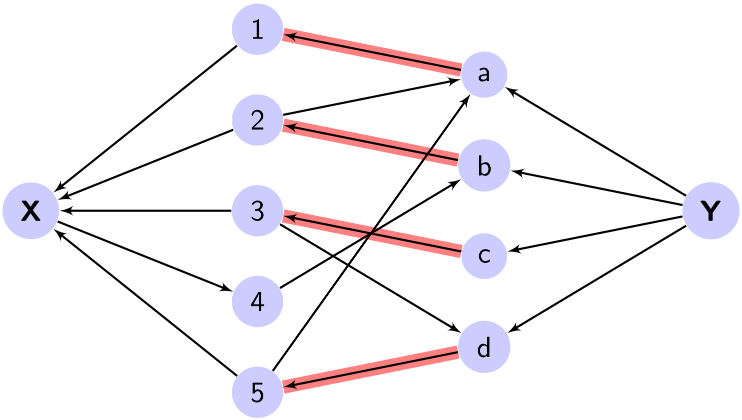
Flow Algorithm

Reverse edges



Flow Algorithm

No more paths found. Matching is reversed edges.



Time Complexity

Finding a path from source to sink using either a DFS or BFS takes $O(E)$ time. For each path found, an incoming edge to the sink is reversed. Since there are only $O(V)$ edges to the sink, at most $O(V)$ paths will be found. Hence the entire algorithm takes $O(VE)$ time, although in practice, it often runs much faster.

Time Complexity

Finding a path from source to sink using either a DFS or BFS takes $O(E)$ time. For each path found, an incoming edge to the sink is reversed. Since there are only $O(V)$ edges to the sink, at most $O(V)$ paths will be found. Hence the entire algorithm takes $O(VE)$ time, although in practice, it often runs much faster.

This approach works for any general unweighted directed graph. We can further simplify it by considering an optimisation which is specific to bipartite graphs. We need not introduce a source or sink node, but rather we can simply go through each node, greedily adding a edge whenever its available whilst recursively trying to reallocate nodes for edges which aren't available.

Alternate Approach

We do the following for each left node i in set U.

- Consider all neighbours j of node i in set V.

Alternate Approach

We do the following for each left node i in set U.

- Consider all neighbours j of node i in set V.
- For each neighbour, try assigning it to node i.

Alternate Approach

We do the following for each left node i in set U.

- Consider all neighbours j of node i in set V.
- For each neighbour, try assigning it to node i.
- If node j has not been assigned to any node, then simply assign it to node i and proceed with the next node $i + 1$.

Alternate Approach

We do the following for each left node i in set U.

- Consider all neighbours j of node i in set V.
- For each neighbour, try assigning it to node i.
- If node j has not been assigned to any node, then simply assign it to node i and proceed with the next node $i + 1$.
- Else, if node j has already been assigned to some other node k, then recursively check whether node k can be assigned to some other node.

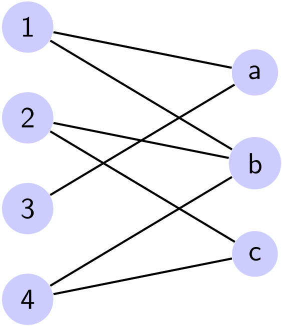
Alternate Approach

We do the following for each left node i in set U.

- Consider all neighbours j of node i in set V.
- For each neighbour, try assigning it to node i.
- If node j has not been assigned to any node, then simply assign it to node i and proceed with the next node $i + 1$.
- Else, if node j has already been assigned to some other node k, then recursively check whether node k can be assigned to some other node.
- To ensure node k doesn't get assigned back to node j, we mark node j as *seen* before making the recursive call.

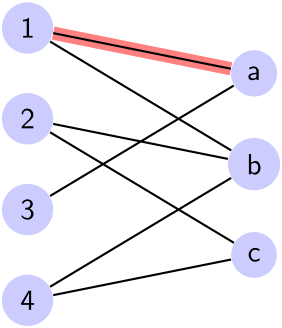
Alternate Example

Given example with 4 left nodes and 3 right nodes:



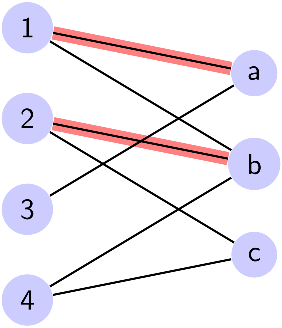
Alternative Example

Start at node 1, start at first neighbour a, pair up (1, a)



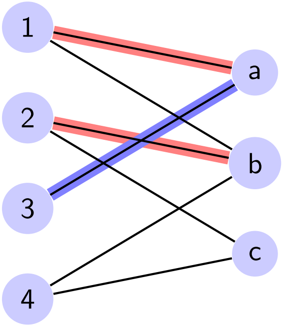
Alternative Example

Start at node 2, start at first neighbour b, pair up (2, b)



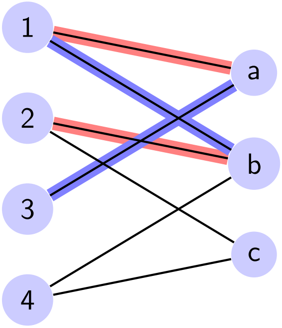
Alternative Example

Start at node 3, start at first neighbour a, node a already taken by node 1, recursively find alternative match for node 1



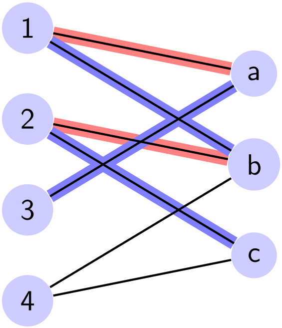
Alternative Example

At node 1, go to next neighbour node b, node b already taken by node 2, recursively find alternative match for node 2.



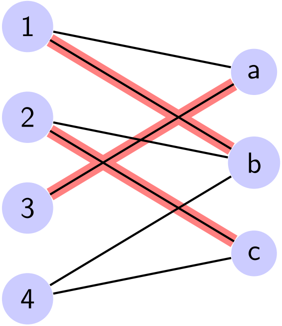
Alternative Example

At node 2, go to next neighbour node c, node c available.



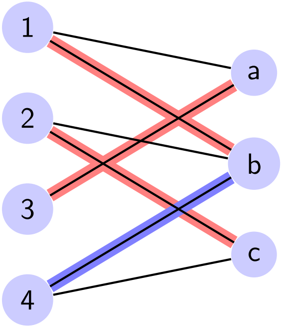
Alternative Example

Pair up (2, c), (1, b), (3, a)



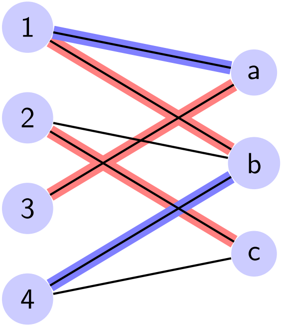
Alternative Example

Start at node 4, start at first neighbour b, node b already taken by node 1, recursively find alternative match for node 1



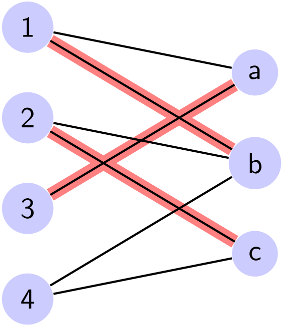
Alternative Example

At node 1, start at first neighbour a, node a already taken by node 3, recursively find alternative match for node 3.



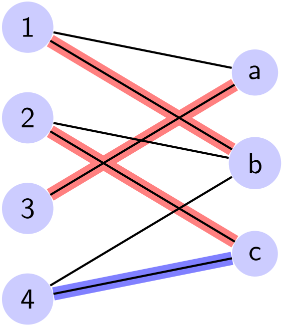
Alternative Example

At node 3, no available neighbours left, recursion failed to provide alternate match.



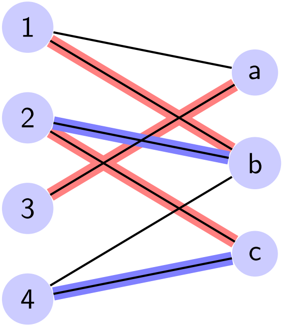
Alternative Example

Back to node 4, try next neighbour node c, node c already taken by node 2, recursively find alternative match for node 2.



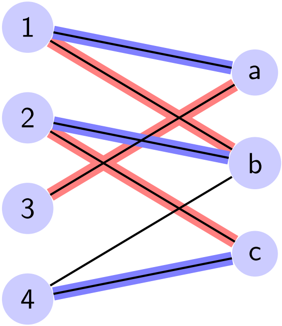
Alternative Example

At node 2, start at first neighbour b, node b already taken by node 1, recursively find alternative match for node 1.



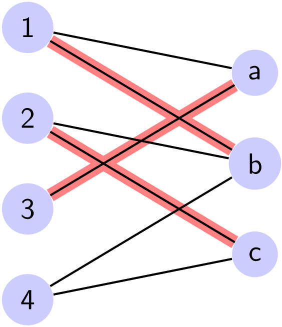
Alternative Example

At node 1, start at first neighbour a, node a already taken by node 3, recursively find alternative match for node 3.



Alternative Example

At node 3, no available neighbours left, recursion failed to provide alternate match. After trying all neighbours of node 4, we conclude node 4 cannot be paired up.



Time Complexity

This alternate solution has the same time complexity as the network flow approach. For each left node i, we could recursively traverse through all edges trying to find a match, taking $O(E)$ time. Doing this for each left node, gives us a total runtime of $O(VE)$.

Time Complexity

This alternate solution has the same time complexity as the network flow approach. For each left node i, we could recursively traverse through all edges trying to find a match, taking $O(E)$ time. Doing this for each left node, gives us a total runtime of $O(VE)$.

This approach may be conceptually harder than the network flow approach, however we can take advantage of recursively calling a `FindMatch` function which results in a lot less code required.

Code


```
def FindMatch(i):
  for all neighbours j of i:
    if not seen[j]:
      seen[j] = True
        if match[j] < 0 or FindMatch(match[j]):
          match[j] = i       ## Pair up (i, j)
          return True
  return False

def BipartiteMatching():
  numMatches = 0
  for all i in |U|:
    seen[] = (False)*|V|
    if FindMatch(i):  numMatches++
```

Example Problem 1

There are M job applicants and N jobs. Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be appointed for only one job. Find an assignment of jobs to applicants in such a way that as many applicants as possible get jobs.

Example Problem 1

There are M job applicants and N jobs. Each applicant has a subset of jobs that he/she is interested in. Each job opening can only accept one applicant and a job applicant can be appointed for only one job. Find an assignment of jobs to applicants in such a way that as many applicants as possible get jobs.

Solution:
This is a simple **direct** implementation of maximal bipartite matching.

Example Problem 2

There are two sets of numbers A and B. Some numbers in set B may be a multiple of certain numbers in set A. Find the least number of numbers that should be removed from either set A or B such that no number in set B is a multiple of some number in set A.

Example Problem 2

There are two sets of numbers A and B. Some numbers in set B may be a multiple of certain numbers in set A. Find the least number of numbers that should be removed from either set A or B such that no number in set B is a multiple of some number in set A.

Solution:
We consider a useful theorem, **Kőnig's theorem**, which states: In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover.

Example Problem 2

There are two sets of numbers A and B. Some numbers in set B may be a multiple of certain numbers in set A. Find the least number of numbers that should be removed from either set A or B such that no number in set B is a multiple of some number in set A.

Solution:

We consider a useful theorem, **Kőnig's theorem**, which states: In any bipartite graph, the number of edges in a maximum matching equals the number of vertices in a minimum vertex cover.

Hence the solution is simply the number obtained when doing a maximal biparite matching.