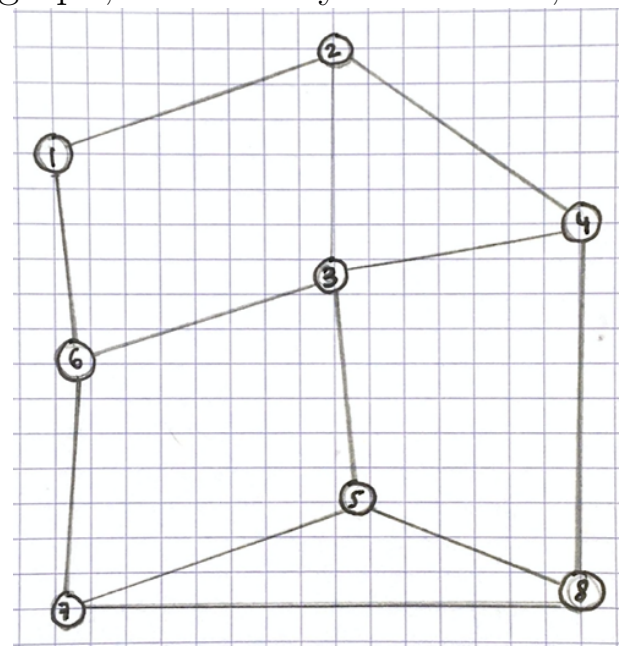
# Lecture Notes on Matchings

Jim Newton

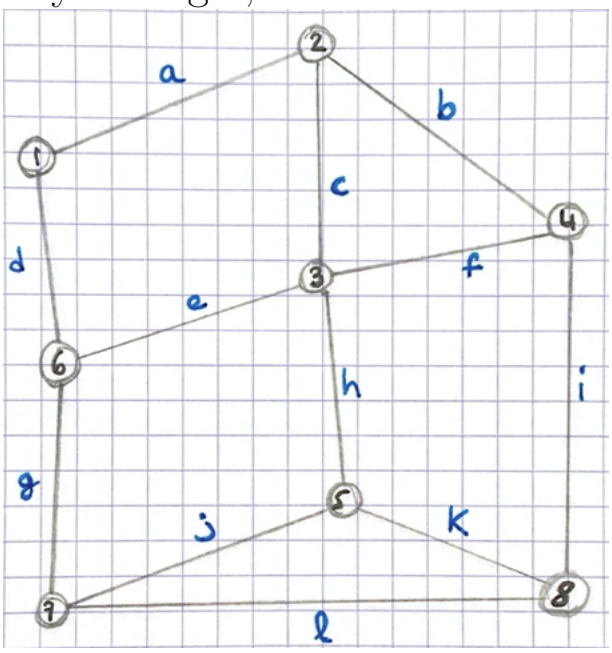
September 22, 2026

## 1 What is a matching

For the next several definitions and examples, we will refer to the following graph, $G$ either by its vertices, $V$ or by its edges, $E$.




Graph with vertices
$V = \{1, 2, 3, 4, 5, 6, 7, 8\}$




Graph with edges
$E = \{a, b, c, d, e, f, g, h, i, j, k, \ell\}$

**Definition 1.1** (incident edges)**.** For graph $(V, E)$, the edges $e_1 = \{v_1, v_2\}, e_2 = \{v_3, v_4\} \in E$ are said to be *incident* if they share a vertex: *i.e.*,

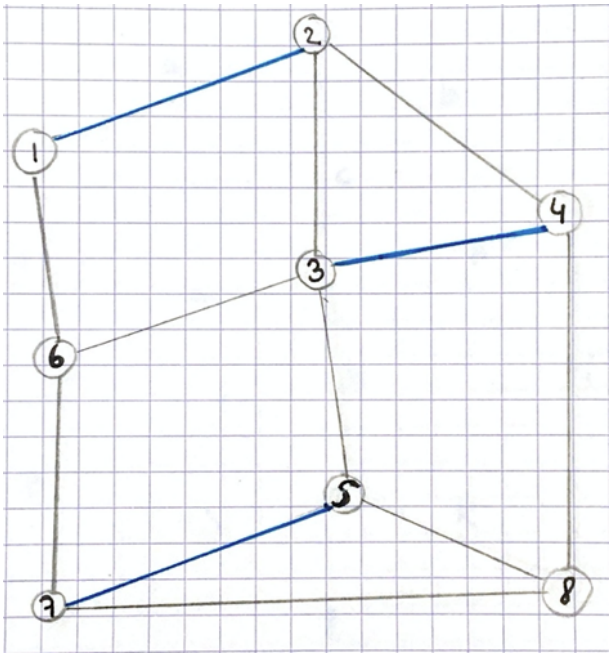
$$e_1 \cap e_2 = \{v1, v2\} \cap \{v3, v4\} \neq \emptyset\,.$$

**Definition 1.2** (independent edges)**.** For graph $(V, E)$, two edges $e_1 = \{v_1, v_2\}, e_2 = \{v_3, v_4\} \in E$ are said to be *independent* if they are not incident: *i.e.*,

$$e_1 \cap e_2 = \{v1, v2\} \cap \{v3, v4\} = \emptyset\,.$$

A set of edges containing having no two incident edges is also called *independent*.

**Definition 1.3** (matching)**.** If $G = (V, E)$ is a simple undirected graph, a *matching*, $M \subseteq E$, is a set of independent edges of $G$.

**Example 1.1.** 

$M_1$ is a matching in this graph.

$$M_1 = \big\{\{1, 2\}, \{3, 4\}, \{5, 7\}\big\}$$

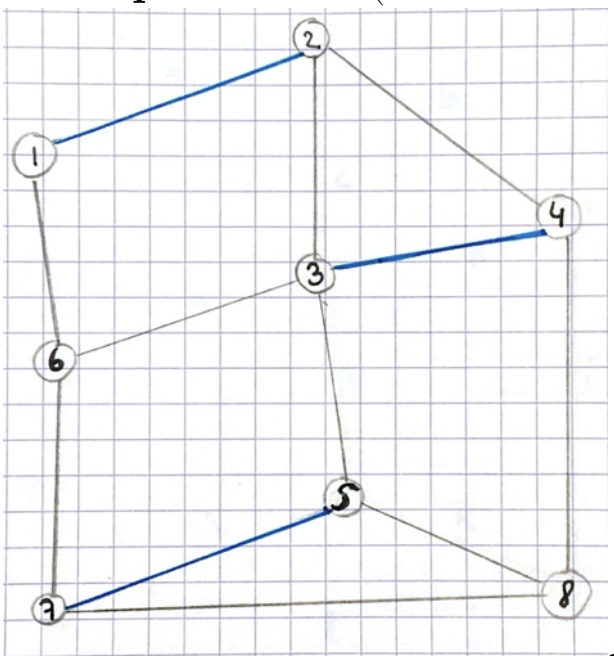
# 2 Maximal and Maximum matchings

The terms *maximal matching* and *maximum matching* are confusing. Be careful with their definitions.

**Definition 2.1** (maximal matching)**.** A matching, $M$, is said to be *maximal*, if there exists no matching $M' \neq M$ for which $M \subset M'$

**Definition 2.2** (maximum matching)**.** A matching, $M$, is said to be *maximum* (or *matching of maximum cardinality*), if there exists no matching $M'$ such that $|M'| > |M|$.

**Example 2.1** (Determine whether a matching is maximal)**.**



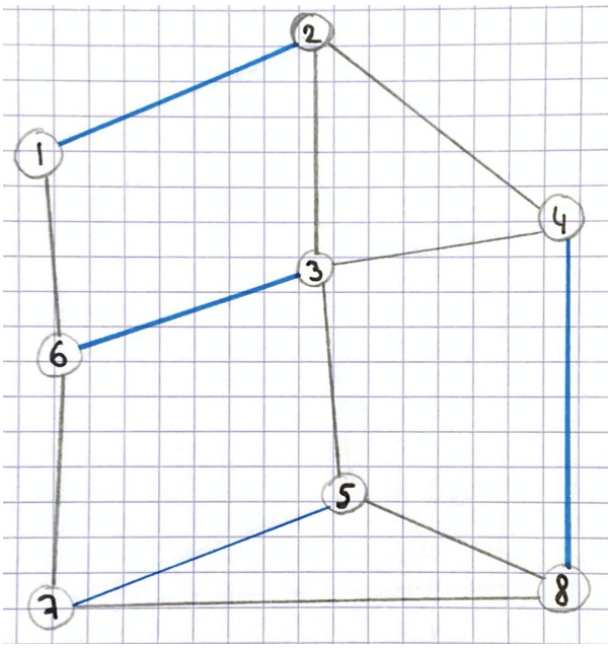

$M_1 = \big\{\{1,2\},\{3,4\},\{5,7\}\big\}$

In the graph, each superset $M'$ of $M_1$, which is a subset of $E$ is not a matching. I.e., no matching different from $M_1$ contains $M_1$. So $M_1$ is maximal.

To see that this is true, just consider any edge of $G$ which is not already in the matching $M_1 = \big\{\{1,2\},\{3,4\},\{5,7\}\big\}$. Those edges are: $\big\{\{1,6\},\{2,3\},\{3,4\},\{3,5\},\{5,8\},\{6,7\},\{7,8\}\big\}$. Each of those edges is disqualified from being in any matching which is a superset of $M_1$, because each of the edges in $E \setminus M_1$ contains a vertex already in $\bigcup M_1$.

**Example 2.2.** $M_4 = \big\{\{1,2\},\{2,4\}\big\}$ is not a matching because $\{1,2\} \cap \{2,4\} = \{2\} \neq \emptyset$.

**Example 2.3.** $M_1 = \big\{\{1,2\},\{3,4\},\{5,7\}\big\}$ is not maximum, because $|M_1| = 3$, but there exists matching $M_3$ of size 4.




$$M_3 = \big\{\{1,2\},\{3,6\},\{4,8\},\{5,7\}\big\}$$
$$|M_3| = 4$$

# 3 Perfect matching

**Definition 3.1** (perfect matching)**.** A matching $M$ of graph $G = (V, E)$, is said to be *perfect*, if $\bigcup M = V$. I.e., every vertex, $v \in V$ is an endpoint of some edge in $M$.

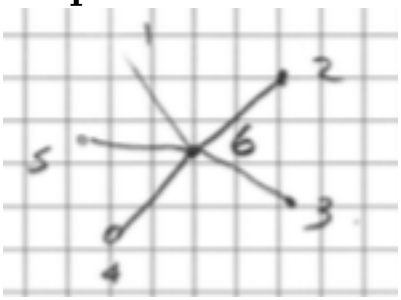
**Example 3.1.** $M_1$ from Example 1.1 is not perfect, because $v = 7$ is not in any element of $M_1$. Recall

$$M_1 = \big\{\{1,2\},\{3,4\},\{5,7\}\big\}$$

**Example 3.2.** $M_3$ from Example 2.3 is a perfect matching because

$$\begin{aligned}\bigcup M_3 &= \{1,2\} \cup \{3,6\} \cup \{4,8\} \cup \{5,7\} \\ &= \{1,2,3,4,5,6,7,8\} \\ &= V\end{aligned}$$

**Example 3.3.** Some maximum matchings are not perfect.




$M_7 = \big\{\{1,6\}\big\}$. $M_7$ is maximum but no edge contains $v = 2$ (for example).

**Claim 3.1.** Every perfect matching is maximum.

**Claim 3.2.** Every maximum matching is maximal.

**Claim 3.3.** No matching, $M$ of graph $G = (V, E)$ contains more than $\lfloor \frac{|V|}{2} \rfloor$ edges. I.e., $|M| \leq \lfloor \frac{|V|}{2} \rfloor$.

**Claim 3.4.** A perfect matching contains exactly $\frac{|V|}{2}$ edges.

**Claim 3.5.** Any matching with $\lfloor \frac{|V|}{2} \rfloor$ edges is maximum.

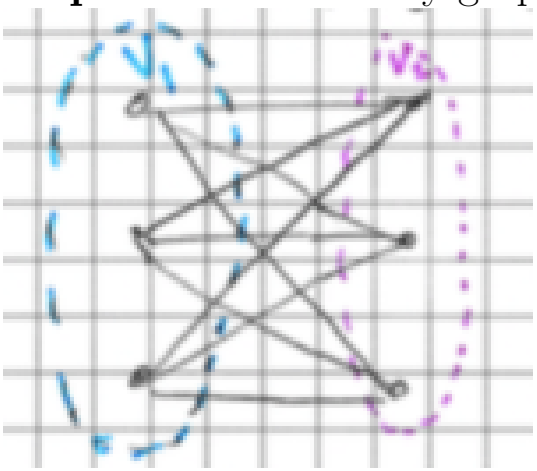
Note that Claim 3.5 is not *if and only if*. Recall Example 3.3, in which $M_7 = \big\{\{1,6\}\big\}$ is maximum, but $|M_7| = 1 < 3 = \lfloor \frac{|V|}{2} \rfloor$

**Definition 3.2** (maximum-weight matching)**.** For a weighted graph, a *maximum-weight matching* is a matching which maximizes the sum of the edge weights.

# 4 Bipartite graphs

**Definition 4.1** (bipartite graph)**.** A *bipartite* graph, $G = (V, E)$, is a graph for which $V$ can be partitioned into $V = V_1 \cup V_2$ where $V_1 \cap V_2 = \emptyset$ (denoted $V = V_1 \sqcup V_2$) and every edge $e \in E$ is of the form $x, y$ with $x \in V_1$ and $y \in V_2$.

**Example 4.1.** The utility graph is bipartite.




**Example 4.2.** Every tree is bipartite.

**Definition 4.2** (tree)**.** (Recall) A *tree* is an undirected acyclic graph.

**Example 4.3.** Let $V_1$ be a set of jobs, each of which requires a different (known) amount of memory. Let $V_2$ be a set of machines, each of which is configured with different amounts of memory. There is an edge $(x, y) \in E$ if job $x \in V_2$ can run on machine $y \in V_2$.

A perfect matching of $F = (V_1 \sqcup V_2, E)$ will specify how to run all the jobs concurrently.

A maximum matching of $F = (V_1 \sqcup V_2, E)$ will specify how to run as many jobs as possible at once.

**Example 4.4.** Let $V_1$ be a set of princesses looking for husbands.

Let $V_2$ be a set of princes looking for wives.

A princesses will marry a prince only if his kingdom is larger than her own.

A prince will marry a princess only if he finds her beautiful.

A perfect matching will determine whether all the princes and princesses can marry.

**Example 4.5.** A multinational army has a supply of tanks and a supply of tank drivers. Each tank needs two drivers. Each driver speaks

one or more languages. Two drivers of the same tank must speak a common language. How many tanks can be operated at once depends on the maximum matching.

# 5 Finding a maximum matching

In the previous sections we discussed *maximal matchings* (Definition 2.1) and *perfect matchings* (Definition 3.1). We can construct simple algorithms for determining whether a given matching is maximal or whether a given matching is perfect. An algorithm to determine whether a given matching is maximal could be based on Example 2.1. An algorithm to determine whether a given matching is perfect could be based directly of Definition 3.1, alternatively such an algorithm could be based off Claims 3.1, 3.2, 3.4, and 3.5.

Unfortunately, none of what we have seen so far gives us a set of necessary and sufficient conditions for determining whether a given matching is *maximum*. Claim 3.5 is sufficient but not necessary, see Example 3.3. The goal is to develop such an algorithm.

**Definition 5.1** (free vertex)**.** Given a graph $G = (V, E)$, and a matching $M \subseteq E$, a vertex is called *free* (or *exposed*, or *unmatched*), if it is not an endpoint of any edge in $M$. Otherwise it is called a *matched* vertex.

Note that by our definition, isolated vertices are free.

**Definition 5.2** (alternating path)**.** Given graph $G = (V, E)$, an *alternating path* with respect to matching, $M$, is a sequence of edges $(e_1, e_2, \ldots, e_k)$ such that no vertex is repeated, and $e_i$ is incident to $e_{i+1}$ for $1 \leq i < k$, and

- either $e_i \in M$ if $i$ is even and $e_j \in M \setminus E$ if $j$ is odd,
- or $e_i \in M$ if $i$ is odd and $e_j \in M \setminus E$ if $j$ is even.

**Definition 5.3** (augmenting path)**.** An *augmenting path* is an alternating path between two free vertices.

## 5.1 Berge's Lemma

Berge's Lemma (1957) states that if $M$ is a matching for a graph, then $M$ is maximum if and only if there is no augmenting path.

Alternatively, there is an augmenting path if and only if $M$ is not maximum.

**Claim 5.1.** An augmenting path has $k$ matched edges and $k+1$ unmatched edges, for some integer, $k$.

**Claim 5.2.** Every augmenting path has an odd number of edges because $k + k + 1$ is odd.

**Claim 5.3.** No other matched edge of a graph intersects an augmented path, for if it did, the path would contain a branch and thus wouldn't be a path.

**Definition 5.4** (set exclusive or (XOR))**.** Given two sets $M$ and $P$, the exclusive-or of the two sets, denoted $M \oplus P$, is defined as follows:

$$M \oplus P = (M \setminus P) \cup (P \setminus M)$$

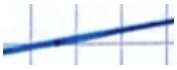
That is the set of elements which are either in $M$ or $P$ but not in both.

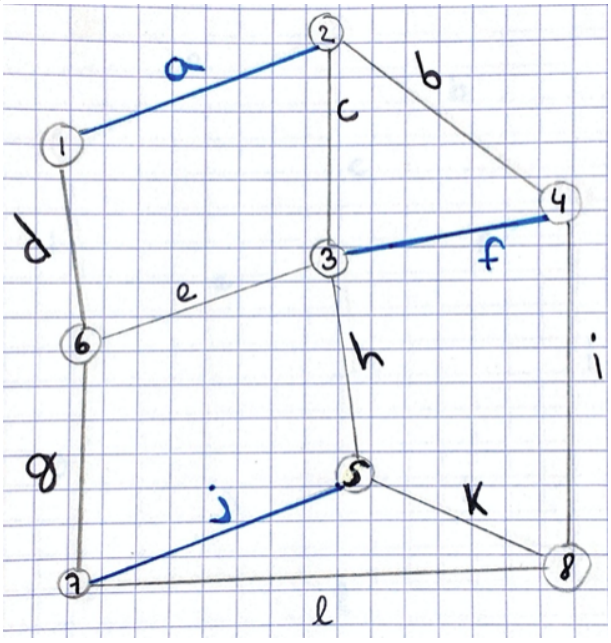
**Claim 5.4.** If $M$ is a matching, and $P$ is the set of edges in an augmenting path, then $M \oplus P$ is a matching exactly one larger than $M$:

$$|M \oplus P| = |M| + 1$$

It is interesting to note that in Claim 5.4, that the integer value of $|M \oplus P|$ does not depend on $|P|$. Consequently, and perhaps counter-intuitively, finding the longest possible alternating path does not aid in

finding a maximum patching, no more than finding the shortest possible alternating path.

**Example 5.1.** We visit again the graph in Example 1.1, but we number the edges $a, b, \ldots, \ell$ and the vertices as in the original example, with the edges in $M$ colored as  .



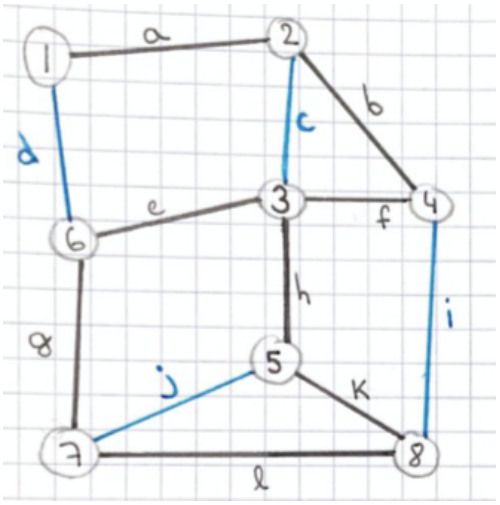

Can we find an augmenting path? Such a path would start and end on free vertices. The only free vertices of $G$ are 6 and 8, so any such path, if it exists, would start and end there. Several such paths exist.

$$P_1 = d\ a\ c\ f\ i \qquad P_3 = e\ f\ i$$
$$P_2 = g\ j\ k \qquad P_4 = d\ a\ b\ f\ h\ j\ \ell$$

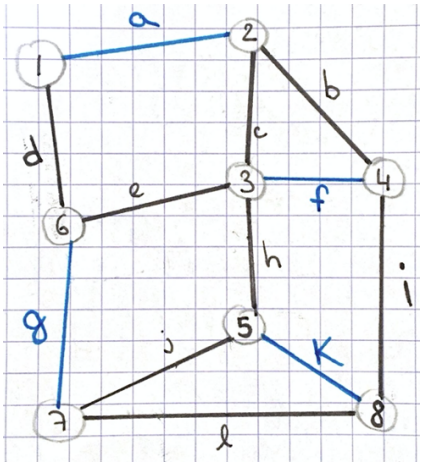
To calculate $M_2$ such that $|M_2| = |M_1| + 1$, we can calculate any of the following.

$$M_2 = M_1 \oplus P_1 \qquad M_2 = M_1 \oplus P_3$$
$$M_2 = M_1 \oplus P_2 \qquad M_2 = M_1 \oplus P_4$$

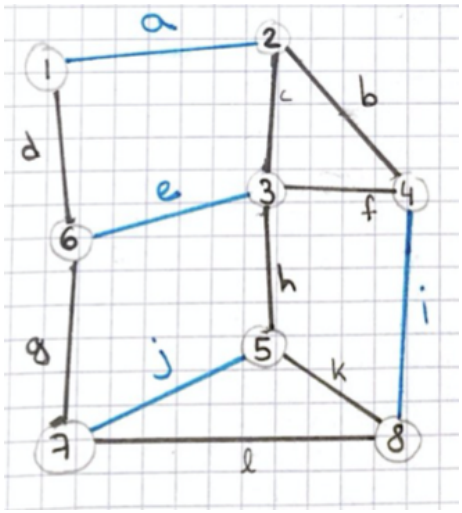
Let's look at each possibility. Since all of these contain 4 edges and $\frac{|V|}{2} = \frac{8}{2} = 4$, then each of the matchings is maximum.



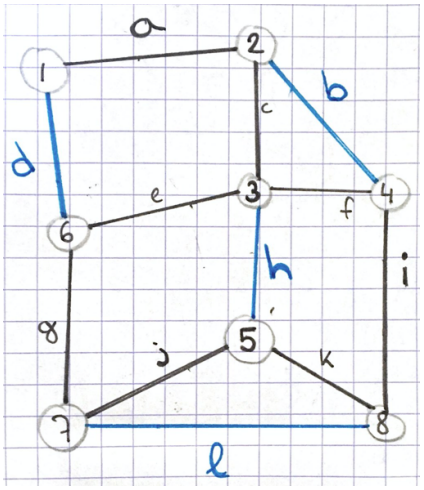

$M_1 \oplus P_1$
$= \{a, f, j\} \oplus \{d, a, c, f, i\}$
$= \{d, c, i, j\}$




$M_1 \oplus P_2$
$= \{a, f, j\} \oplus \{g, j, k\}$
$= \{a, f, g, k\}$




$M_1 \oplus P_3$
$= \{a, f, j\} \oplus \{e, f, i\}$
$= \{a, j, e, i\}$




$M_1 \oplus P_4$
$= \{a, f, j\} \oplus \{d, a, b, f, h, j, \ell\}$
$= \{d, b, h, \ell\}$

Given an augmenting path, such as $P = \{d, a, c, f, i\}$, we may equivalently reverse the *roles* (matching vs. non-matching) of its edges. For example. We can reverse the roles of $P$ to obtain $P_{reversed} = \{d, a, c, f, i\}$. Partition $P_1$ into $P_1 = P_{in} \sqcup P_{out}$, forming two sets $P_{in} = \{d, c, i\}$ and

$P_{out} = \{a, f\}$.

$$\begin{aligned} P_{in} \cup (M_1 \setminus P_{out}) &= \{d, c, i\} \cup \big(\{a, f, j\} \setminus \{a, f\}\big) \\ &= \{d, c, i\} \cup \{j\} \\ &= \{d, c, i, j\} \\ &= M_1 \oplus P_1 \end{aligned}$$

Not all alternating paths have the same length. For example, the lengths $|P_1| = 5$, $|P_2| = 3$, $|P_3| = 3$, and $|P_4| = 7$ are different. If we want to use a breadth-first-search based algorithm, then we can abort the descent on finding the first augmenting path, which will be one of minimum length. *I.e.*, if finding long augmenting paths is more difficult than finding short ones, then we would like to find a *short* augmenting path.

**Claim 5.5.** Also, notice that the algorithm described above does not specify how to find an augmenting path, but rather given such a path, how to improve the matching.

We would like an algorithm which not only finds an augmenting paths, but also finds a shortest augmenting path.

**Claim 5.6.** In general there may be more than two free vertices. There may not be an augmenting path between certain pairs of free vertices. It is not clear which pairs of vertices to search for augmenting paths between. However, we can only hope to find such a path if it has odd length. See Claim 5.2.

**Claim 5.7.** Improving a matching via an augmented path increases the size of the matching by exactly one.

**Claim 5.8.** Finding a maximum matching finds a perfect matching if a perfect matching exists, because a perfect matching contains exactly $\frac{|V|}{2}$ edges, and that is the maximum possible size of a matching.

## 5.2 Maximum Matching Algorithm

Edmond's Blossom Algorithm (1961) provides a solution which has complexity $O(|E||V|^2)$. We will instead look at a BFS-type search algorithm.

Algorithm 1 is a simple recursive procedure to find a maximum matching, which continues to extend the given matching by one edge each time as long as it is possible to find an augmenting path relative to the currently computed matching.

The procedure depends on `find-augmenting-path-or-none` which is more complicated. We discuss `find-augmenting-path-or-none` in Section 7.

```
1.1 Function find-maximum-matching(G, M)
        Input  : G a simple graph
        Input  : M a possibly empty matching
1.2     p ← find-augmenting-path-or-none(G,M)
1.3     if p is an augmenting path then
                                  // recur with matching, 1 longer than M
1.4       return find-maximum-matching(G, M ⊕ p)
1.5     else
1.6       return M
```

**Algorithm 1:** Function to find a maximum matching.

# 6 Immutable graph search

The procedure in Algorithm 2 computes a set of path extensions. If the given path has length $k$, then the set returned is a possibly empty set of paths, each of length $k + 1$. The set will be empty if all potential path extensions would be self-looping.

2.1 **Function** *extend-path-by-1*$(adj, k, p)$

**Input :** $adj$ adjacency list of simple graph

**Input :** $k$ an integer designing length

**Input :** $p$ a path of length $k$

**Result:** Takes a $k$-length non-looping path and computes a set of $k+1$ length non-looping paths with the given path as a tail.

2.2 **return** $\{u :: p \mid u \in adj[p_0] \wedge u \notin p\}$

**Algorithm 2:** Set comprehension to generate a possibly empty set of paths of length 1 greater than the given path.

The procedure uses the notation $u :: p$. This is intended to be a prepend operation. In many programming languages a non-destructive prepend is very cheap, because the list data structure typically shares tails with other lists. However, in Python, using either array/list `[u] + p` or `p + [u]` it is expensive. Why? Because it must copy the entire list every time it is called.

Even if it is inefficient in terms of memory, in Python you can use a list comprehension for concise code which looks similar to the analogous set theoretical expression. The function looks something like the following.

```
def extend_path_by_1(adj,k,p):
  return [[u] + p for u in adj[p[0]]
              if u not in p]
```

You will notice that $k$ is unused, and could be eliminated from these procedures; however, we will make use of it later. It is included for the example to better understand what is happening in terms of path length.

Python lists (arrays) are indexed by 0, so we have `p[0]` in the Python code corresponding to $p_0$ in the pseudo code of Algorithm 2.

Given function `extend-path-by-1` we can implement `extend-paths-by-1` as shown in Algorithm 3.

```
3.1 Function extend-paths-by-1(adj, k, paths_old)
        Input  : adj adjacency list of simple graph
        Input  : k an integer designing length
        Input  : paths_old set of paths all of length k
        Result: Takes a set of k-length non-looping paths and
                computes a set of k + 1 length non-looping paths.
                All paths are in reverse order with the most
                recently added vertex at the beginning.
3.2     for p ∈ paths_old do
3.3         for q ∈ extend-path-by-1(adj, k, p) do
3.4             collect(q)
3.5     return collected paths
```

**Algorithm 3:** BFS graph descent with immutable data structure

In Python, using list comprehensions, Algorithm 3 looks like the following.

```
def extend_paths_by_1(adj,k,paths):
  return [q for p in paths
            for q in extend_path_by_1(adj,k,p)]
```

At the rest of loosing readability, you may of course in-line `extend-path-by-1` into extend-paths-by-1.

# 7 Finding an augmenting path

In this section we will develop an algorithm for finding an augmenting path. Such a function can be used in Algorithm 1 to find a maximum matching. We will first present a trivial solution in the form of Algorithm 4 and thereafter improve the solution.

## 7.1 Naïve solution to find augmenting path

If a graph has an edge which is incident to two free non-adjacent vertices, then that edge gives us an alternating path trivially. *I.e.*, if $\{u, v\}$ is an edge and both $u$ and $v$ are free vertices, then $[u, v]$ is an alternating path (see Definition 5.2).

Otherwise, we may construct a naïve algorithm based on Algorithm 3. Algorithm 4 simply searches longer and longer paths until either there are no more paths, in which case it returns $None$, or it finds an augmenting path and returns it.

```
4.1  Function naive-find-augmenting-path-or-none(adj, E, M)
         Input  : adj adjacency list of simple graph
         Input  : E set of edges
         Input  : M a matching

          // Find set of all vertex-length=2 paths ENDING at a free vertex
4.2      paths ← {[v, u] | u ∈ free, {u, v} ∈ E}
4.3      k ← 1                    // paths start with 2 vertices, ie., k=1 edge
4.4      while paths ≠ ∅ do
4.5          for p ∈ paths do
4.6              if p is an augmenting path then
4.7                  return p
4.8          paths ← extend-paths-by-1(adj, k, paths)
4.9          k ← k + 1
4.10     return None
```

**Algorithm 4:** Naïve function to find augmented path

Take a look line 4.2 of Algorithm 4 which is repeated as line 5.5 of Algorithm 5.

$$paths \leftarrow \{[v, u] \mid u \in free, \{u, v\} \in E\}$$

This set comprehension computes all the 2-vertex paths terminating on a free vertex. For example, if $v_1$ is a free vertex and $v_1$ is adjacent to vertices

$v_2$ and $v_3$, then we consider the edges $\{v_1, v_2\}$ and $\{v_1, v_3\}$, which includes $[v_2, v_1]$ and $[v_3, v_1]$. If in addition $v_2$ is a free vertex, then $[v_1, v_2]$ is also included. In Python this list comprehension will look something like the following:

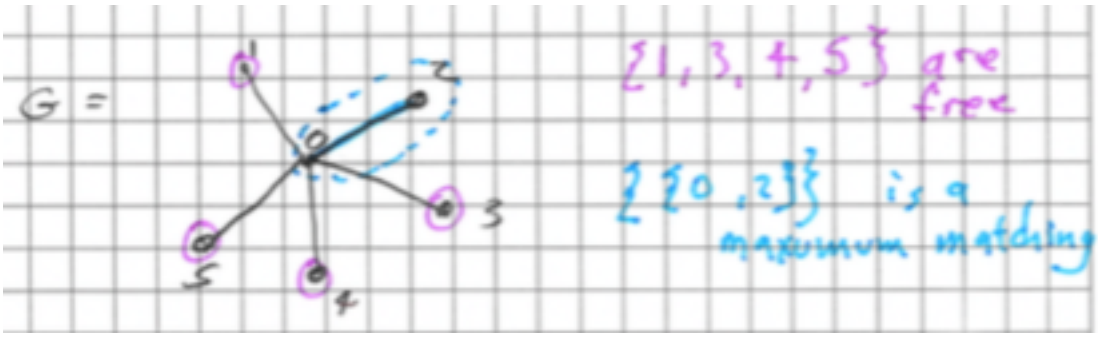
```
paths = [[v,u] for u in free for v in adj[u]]
```

The algorithm isn't completely naïve, as it only searches paths which end on a free vertex. Also if any path, $[v, u]$, in the initial set of paths constructed on line 4.2, is such that $u$ and $v$ are both free vertices, then the test on line 4.6 will detect it and return it without ever calling `extend-paths-by-1`.

## 7.2 Improving the augmented path search

We can improve on Algorithm 4 in several ways. Algorithm 5 presents a straightforward algorithm to compute an augmenting path if one exists. The algorithm is an improvement over Algorithm 4 and is inspired by Algorithms 2 and 3.

The procedure is shown in Algorithm 5. First, line 5.3, a simple optimization: if there fail to be two free non-isolated vertices, with respect to a matching $M$, then $M$ is a maximum matching. Thus there is no need to search for an augmenting path unless we can find 2 free non-isolated vertices. So if we hope to find a maximum matching, we must thus find (at least) two free vertices. However, given two free vertices there is no guarantee that an augmenting path exists. For example, in the following graph, any non-trivial matching has exactly one edge, leaving four free vertices.




There is a second way we can improve on the naïve Algorithm 4. We notice that in Algorithm 4, we generate all alternating paths of length $k$, and then examine them after their construction to decide whether any happens to be augmenting. To check whether a path is augmenting, we need to check

1. whether the first vertex is free,

2. whether the final vertex is free, and

3. whether the path is alternating.

However, the way we are generating the paths, the final vertex is guaranteed to be free. We have replaced the call to `extend-paths-by-1` with a call to `extend-alternating-paths` (yet to be implemented), thus generating only alternating paths (by construction). The only condition remaining is to check whether the initial ($0^{th}$) vertex is free, which we do on line 5.10.

Additionally, if the path has an even number of edges, *i.e.* if it has an odd number of vertices, then its $0^{th}$ point is necessarily not free, so no need to check it. In Algorithm 5, $k$, is number of edges in the paths; so it suffices to check $p_0 \in free$ only when $k$ is odd, line 5.8.

```
5.1 Function find-augmenting-path-or-none(adj, E, M)
        Input  : adj adjacency list of simple graph
        Input  : E set of edges
        Input  : M a matching
5.2     free ← generate-free-vertices()
5.3     if |free| < 2 then
5.4         return None
        // Find set of all vertex-length=2 paths ending at a free vertex
5.5     paths ← {[v, u] | u ∈ free, {u, v} ∈ E}
5.6     k ← 1                                          // how many edges?
5.7     while paths ≠ ∅ do
5.8         if odd(k) then
5.9             for p ∈ paths do
5.10                if p_0 ∈ free then
5.11                    return p
5.12        paths ←
              extend-alternating-paths(adj, M, k, free, paths)
5.13        k ← k + 1
5.14    return None
```

**Algorithm 5:** Function to find an augmenting path if one exists.

## 7.3 Generating alternating paths by construction

Algorithm 6 references the function, `extend-alternating-paths`, which is implemented as Algorithm 6. The function, `extend-alternating-paths`, takes a set of alternating paths, all of the same length, and produces one of two possible results.

1. If one of the given alternating paths can be extended one-step to form an augmenting path, then return a singleton set of that augmenting path.

2. Otherwise the function returns a possibly empty set consisting of all possible non-looping alternating paths, which are one-step extensions of one of the given alternating paths, guaranteed to be non-looping.

```
6.1  Function extend-alternating-paths(adj, M, k, free, paths)
         Input  : adj adjacency list of simple graph
         Input  : M a matching
         Input  : k number of edges in each path in paths. Too
                  produce paths with k + 1 edges.
         Input  : free set of free vertices
         Input  : paths set of alternating paths, each a vertex array
         Result: Returns a possibly empty set of alternating paths,
                 each of length k + 1

6.2      for p ∈ paths do
6.3          for u ∈ adj[p_0] do
6.4              if u ∉ p then // avoid looping paths
6.5                  if odd(k) ∧ {u, p_0} ∈ M then
6.6                      collect [u, p_0, p_1, ...p_k]          // matching edge
6.7                  else if even(k) ∧ {u, p_0} ∉ M then
6.8                      p' ← [u, p_0, p_1, ...p_k]             // non-matching edge
6.9                      if u ∈ free then // thus p is augmenting.
6.10                         return {p'}     // return singleton set of p'.
6.11                     else
6.12                         collect p'
6.13     return collected paths
```

**Algorithm 6:** Function which extends a set of $k$-edge-length alternating paths to a possibly empty set of $k+1$ edge-length alternating paths.