Algorithmic Graph Theory Notes

Avi Semler

2023

Contents

List of Algorithms

1 Basic definitions

1.1 Graphs

An (simple undirected graph) is a pair $G = (V, E)$ with $E \subseteq \binom{V}{2}$. A directed (simple) graph has $E \subseteq V \times V$ instead. A multigraph has E as a multiset of two-element multisets of vertices.

1.2 Terminology

Edges are *incident* to vertices, vertices are *incident* on edges, vertices are *adjacent* to vertices (and edges to edges). Directed edges are *consecutive* if the head of the first one coincides with the tail of the second.

A *path* is a sequence of adjacent vertices, and its length is the number of edges between them. A path is *simple* if the vertices are distinct, and a *cycle* if the start and end vertices coincide.

We define an equivalence relation of (strong) *connectedness* for vertices: requiring a path between them (in both directions if the graph is directed). The equivalence classes are the (strongly) *connected components*. If there is only one connected component, then G is (strongly) *connected*.

A subgraph is spanning if it contains all vertices. It is a tree if it is connected and contains no cycles (acyclic). A spanning tree is a subset $T \subset V$ if it is both spanning and the subgraph it induces is a tree.

2 Exploration

These algorithms list all vertices of the graph that can be reached from some vertex s.

2.1 Breadth-first search

Breadth-first search explores the graph in "layers" of distance from starting vertex s and builds a tree.

Algorithm 2.1 Breadth-first search

```
Let R be a queue
Enqueue (s, s) to R
Initialise T := {}
while R ≠ ∅ do
    Let (x, y) = R.dequeue()
    if y is unvisited then
        visit y
        Add edge (x, y) to T
        for each neighbour n of y do
            if n is unvisited then
                Enqueue (y, n) to R
            end if
        end for
    end if
end while
Remove the self-loop (s, s) and return T
```

The tree T once execution has finished is called the BFS tree.

2.2 BFS tree properties

Lemma 2.1. *Nodes are added to BFS tree T in non-decreasing order of distance in T to s.*

Proof. Assume otherwise, and that v' is the first node for which the propery fails, because there is already v at a distance ℓ and v' is at distance $\ell' < \ell$ from s. Let u, u' denote their predecessors in the tree, necessarily at distances $\ell - 1$ and $\ell' - 1$.

But we know that u' is visited first (because v' was the minimal counterexample), and therefore (u', v') is added to the queue before (u, v) ↯ □

Theorem 2.2. *The BFS tree T contains a shortest (s, v)-path in G for all v reachable from s.*

Proof. Assume otherwise, and let u be the minimal counterexample - the vertex closest to s in G which does not have a shortest path in the tree. Then, let u' be the predecessor of u in some shortest (s, u)-path and let x be the parent of u in T.
By assumption of minimality, a shortest (s, u')-path is in T. So we must have that the length of the (s, x)-path in T is greater than the distance from s to u' (otherwise u *would* have a shortest path in T!). But by theorem 2.1, the edge (u', u) will be added to the tree before the edge (x, u) - this is a contradiction as we now see that u cannot have been visited via x! □

Theorem 2.3. *Graph edges that are not in the BFS tree connect successive layers or vertices from the same layer.*

Proof. Assume that $\{u, v\}$ is a non-tree edge with $u \in L_i$ and $v \in L_j$ with $i < j - 1 \iff i + 1 < j$ (i.e. non-successive layers). By previous theorem, a shortest (s, u)-path has length i and a shortest (s, v)-path has length j. But there is also an edge $\{u, v\}$. But then we get that the shortest (s, v)-path has length at most $i + 1$, because we can go via u. This then implies that $j \leq i + 1$ ↯ □

2.3 DFS

Uses a stack (can be implemented with recursion) instead of a queue.
The DFS finishing number of a vertex records the time that the recursive call initiated when visiting a vertex finishes. These numbers $N[v]$ are all distinct

Lemma 2.4 (Finishing times decrease moving down tree)**.** *If u is the parent of v then $N[u] > N[v]$.*

Proof. The DFS procedure called for u then calls itself for v as the edge connecting u to v must have been in the tree. So it can only finish for u after it has finished for v. □

Lemma 2.5. *Let G be a directed graph. If (u, v) is not used in the DFS-forest then either $N[u] > N[v]$, or u is the ancestor of v in the forest or vice versa.*

Proof. Assume for contradiction that there are vertices u and v with no ancestor relationship and there exists an edge (u, v) with $N[u] < N[v]$.
The only way that (u, v) could end up not being in the forest is if at the point that the edge is considered, v is already visited. At that point in time u has not finished its DFS call, implying that v hasn't finished either (because it has a lower finishing time). So v must be *active* (visited but not finished). Since u is also active, this contradicts the assumption that they do not have an ancestor/descendant relationship (all active nodes do!). □

In a tree in the DFS forest (the union of all trees produced when calling DFS multiple times) there are three types of non-tree edges:

- Edges that connect ancestor to descendant: forward
- Vice versa: backward
- Between "cousins": cross edge

3 Testing bipartiteness

G undirected is bipartite if $\exists V_1, V_2$ that partition V and $\forall e \in E$, $|e \cap V_1| = 1 = |e \cap V_2|$.
For a directed graph, we instead require $\forall e \in E, e \in (V_1 \times V_2) \cup (V_2 \times V_1)$.

3.1 Characterisation

Theorem 3.1. *G is bipartite if and only if it has no odd cycles.*

Proof. Assume G has an odd cycle $(v_1, \ldots, v_n)$. Then clearly, v_1 and v_n belong to different partitions. But we can also prove that all odd-labelled vertices are in the same partition as v_1! ↯
Conversely, assume that any cycles are even. Then, apply the algorithm below and the paritions are the even and odd layers.

□

3.2 Algorithm

Run BFS on each connected component. Then:

Theorem 3.2. *G is bipartite if and only if there are no edges between vertices on the same level.*

Proof. If there is no such edge, we can partition into the even and odd layers so G is bipartite.
Conversely, if there is such an edge (u, v) then there is an odd cycle: $s, \ldots, u, v, \ldots, s$. □

4 Representation

- *Edge list* maintains a list of edges. Takes $O(m)$ space, but is not well suited for most graph operations
- *Adjacency matrix* uses an $n \times n$ matrix A where the value of the entry a_{ij} encodes the presence of a (directed) edge connecting vertex i to vertex j. This is wasteful for sparse graphs
- *Adjacency list* stores a linked list for every vertex, specifying which other vertices it connects to. Takes $O(m + n)$ space - this is the generally assumed representation.

Theorem 4.1 (Lower bound for determining bipartiteness with adjacency matrix)**.** *Any algorithm that takes determines if a graph G is bipartite using its adjacency matrix has running times $\Omega(n^2)$.*

Proof. Let ALG be such an algorithm. Let L be the number of times it accesses the matrix when the input is the star graph on vertices $[n]$ with 1 as the centre vertex (ALG must return TRUE as this is indeed bipartite).
Now, assume that G makes fewer than $\binom{n-1}{2}$ accesses to the matrix. We can then construct a non-bipartite graph G' that ALG cannot distinguish from G as follows:
There must be two vertices of the star that are not the centre that do not have the entries of the matrix that they index accessed (because there are $\binom{n-1}{2}$ such pairs). So add an edge between them to form G'.

G' now contains an odd cycle but ALG still outputs TRUE, so we can conclude that in fact no such algorithm can exist.
Hence all algorithms access the matrix at least $\binom{n-1}{2}$ times. □

5 Directed acyclic graphs

Definition 5.1 (Topological sort)**.** *A map $\phi : V \to [n]$ is a topological sort if for all edges (u, v) we have $\phi(u) < \phi(v)$.*

Theorem 5.2. *A directed graph has a topological sort iff it is acyclic.*

Proof. Suppose there is a topological sort ϕ and a cycle. Then we can walk around the cycle with ϕ always increasing which becomes a contradiction as soon as we repeat a vertex.
Conversely, suppose the graph is acyclic. There exists a vertex of degree 0 (otherwise we could keep walking and find a cycle). So number this vertex n, delete it, and number the rest of the graph recusively. □

This proof is constructive and the algorithm it suggests can be made to run in $O(m+n)$ time. There is an even simpler algorithm for finding a topological sort:

Theorem 5.3. *Let N be an array of DFS finishing times. Then if ϕ maps $v \mapsto n - N[v] + 1$, ϕ is a topological sort.*

Proof. If (u, v) is a tree edge, we know that $N[u] > N[v]$ so the property holds.
If it is not a tree edge, we know that it is either a forwards or cross edge (otherwise we would have a cycle if it was backward). In either case it is true that $N[u] > N[v]$ because we can exclude the case of them having an ancestor relationship if cross or forward.

□

6 Connected components

Definition 6.1. *A connected component of a directed graph is an equivalence class of the connectivity relation (where vertices are related if there is a path between them in both directions).*

7 Biconnected components

Biconnected if still connected after deletion of any vertex. A vertex whose removal disconnects is called a cut-vertex of articulation point.
A maximal biconnected subgraph is a biconnected component. Not necessarily disjoint! The cut vertices are precisely those that lie in intersections.
We can define an equivalnce relation on the edges s.t. the equivlance classes are the edge sets of the biconnected components. Edges are related if they are part of the same simple cycle. The proof of transitivity involves composing the cycles.
can find them with DFS tree. WLOG assume connected. leaves can't be articularion points: the tree does not become disconnected upon removal *even considering just tree edeges*!

if the root has only one child, it's not an articulation point. in every other case, there could be tree edges.
Internal vertices are aritculation points iff it has a child such that no edges connnect from its subtree to an ancestor of the original node.
The low point of a node v is the lowest level among the neighbours of the nodes in subtree T_v.

8 Colourings

8.1 Definitions

We use C as a set of colours, usually natural numbers. A (proper) vertex-colouring is a function $c : V \to C$ such that $c(u) \neq c(v)$ for all u, v adjacent. (Proper if c is surjective).
The chromatic number $\chi(G)$ is definted as the smallest number k for which there exists a k-colouring.

1. The chomatic number of a tree is 2
2. K_n has chromatic number n
3. If G' is a subgraph, then $\chi(G') \leq \chi(G)$

Clique number is the cardinality of the largest subgraph which is complete.

8.2 Algorithms

A simple approach is to colour greedily: use the smallest colour that none of the neighbours have already.

Lemma 8.1. *Greedy colouring uses at most $\Delta(G) + 1$ colours where $\Delta(G)$ denotes the maximum degree in G.*

Proof. Every time we seek to colour a vertex, we will have at least one colour remaining if $|C| = \Delta(G) + 1$. □

Verifying if the chromatic number is k is NP-complete.

9 Planarity

A graph is planar if it can be drawn in $\mathbb{R}^2$ with every vertex as a point and each edge a continuous curve such that no edges intersect. Faces are the connected components of $\mathbb{R}^2$ after drawing the graph.

1. Trees are planar
2. Cycles are planar
3. Every complete graph up to K_4 is planar

Theorem 9.1 (Euler's formula)**.** *For G connected and planar: $|V| + |F| = 2 + |E|$, where F is the set of faces. The number of faces is an invariant of all plane embeddings of G.*

Proof. By induction on the number of edges.
If G is acyclic, then $|F| = 1$ and the theorem holds since G is a tree and $|E| = |V| - 1$.
If instead there is a cycle, let e be an edge in the cycle. When we delete e we get one fewer faces. By induction hypothesis the formula still holds. □

Note that Euler's formula as stated does not hold for non-connected graphs.

Theorem 9.2 (Generalised Euler's formula)**.** $|V| + |F| = 1 + cc_G + |E|$ *where* cc_G *is the number of connected components.*

Planar graphs have their number of edges linear in the number of vertices (sparse):

Theorem 9.3 (Sparsity of planar graphs)**.** *For G simple connected planar with $n > 2$ it holds that $|E| \leq 3n - 6$.*

Proof. Every face is bounded by at least 3 edges (counting every time an edge 'touches' a face, so holds for a path on $n > 2$ vertices too).
Every edge also bounds as most 2 distinct faces (perhaps just one face twice).
For $f \in F$, let $m(f) = \#$edges that bound f. Now:

$$\sum_{f \in F} m(f) \geq 3|F|$$

and, counting from the perspective of the edges:

$$\sum_{f \in F} m(f) \leq 2|E|$$

implying that $2|E| \geq 3|F|$. Now apply Euler's formula:
$|E| = |V| + |F| - 2 \implies 3|E| = 3|V| + 3|F| - 6 \leq 3|V| + 2|E| - 6$ and the conclusion follows. □

By this theorem, we can see that K_5 has too many edges and it not planar.

Lemma 9.4. *Every simple planar graph has a vertex of degree at most 5.*

Proof. Assume otherwise and □

Corollary 9.5. *Every planar graph is 6 colourable.*

Proof. By induction: 6-colour the rest of the graph (apart from a vertex of degree ≤ 5) and then extend the colouring by finding a colour for that vertex (possible cos it only has 5 neightbours at most). □

The proof is constructive and provides an algorithm to compute the colouring in linear time.

Lemma 9.6. *Every simple graph is 5-colourable.*

Proof. By induction.

If G has a vertex of degree ≤ 4, done.

If not: there is one with 5 neighbours. Colour the rest. The hard case here is if the vertex has 5 neighbours with distinct colours. WLOG they are coloured in clockwise order, and label the neightbours $u_1, \ldots$.

Consider the subgraph induced by vertices with colours 1,3. If u_1 and u_3 are disconnected, swap the colours in one of the components. And have one colour left over.

If instead they are connected: We know that u_2 and u_4 have disconnected subgraphs when inducing on their colour, by planarity! So apply the same argument to them. □

And famously all planar graphs are 4-colourable.

The dual graph has the faces as its edges are where faces share a boundary edge. Still planar.

There exists a linear time algorithm for testing planarity.

Theorem 9.7 (Wagner's theorem)**.** *A graph is planar iff it has no minor isomporphic to K_5 or $K_{3,3}$.*

Proof. $\implies$

$\impliedby$ □

10 MSTs

10.1 Meta-algorithm

The meta-algorithm builds a tree edge by edge until there is a tree, by either including an edge that is "blue" or excluding one that is "red".

Definition 10.1 (Red and blue rules)**.** *An edge $e = \{u, w\}$ can be coloured **blue** if there exists a parition U, W of V with $u \in U, w \in W$, no blue edge connecting U and W, and e is the edge of least weight among those that connect U and W and are not red.*

*An edge e can be coloured **red** if there exists a simple cycle that does not already contain a red edge and e is of maximum weight among the uncoloured edges in the cycle.*

Formally, the algorithm can be stated as follows:

Algorithm 10.1 Meta-algorithm for MSTs

```
while there are uncoloured edges do
    apply either blue or red rule
end while
return tree formed by blue edges
```

We want to show that the algorithm will always return an MST.

Lemma 10.2 (Colouring invariant)**.** *At each stage of the while loop, there exists an MST that contains all blue edges and no red edges.*

Proof. On the first iteration, there are no coloured edges so the invariant is vacuously true.

Suppose that the invariant is satisfied before a blue edge e is added. Let T be the MST that satisfied the invariant. If $e \in T$, then T is the tree that satisfies the invariant after adding e too. So assume $e \notin T$, and we will construct an MST including it.

Let U, W be a partition that causes e to be coloured blue. There must exist a path in T that connects the endpoints of e by connectivity of trees. One edge in this path must connect a vertex from U to a vertex in W (otherwise the path would stay in the same parition). Call this edge e'. e' is not blue (because e was coloured blue), and it is also not red (it is in T). So we know $w(e') \leq w(e)$. So we can get a new MST that satisfies the invariant by removing e' and adding e.

Suppose that the invariant is satisfied before a red edge e is added. Let T be as before - the MST that satisfied the invariant. If $e \notin T$, then T is the MST that will also satisfy the invariant after e is coloured red. So assume $e \in T$, and we will construct a MST excluding it.

Let V_1, V_2 be the vertex sets of the two subtrees that we obtain after deleting e from T. Since e is red, there exists a cycle C containing e and other edges with weights $\leq w(e)$. There must be at least one other edge in C besides e that connects V_1 to V_2 - call this edge e'. So add e' and remove e to get the desired MST.

□

Corollary 10.3. *The algorithm is never unable to colour edges if there are still uncoloured edges left.*

Proof. Let e be an uncoloured edge. Due to the invariant, the blue edges form a spanning forest (including some trees of only one vertex).

If both endpoints of e are in one blue tree of the forest, then e can be coloured red as it is part of a blue cycle.

If e's endpoints are in different blue trees, then there is a partition with no blue edges that the blue rule can be applied to.

□

10.2 Kruskal's algorithm

Algorithm 10.2 Kruskal's algorithm

```
all edges begin uncoloured
sort edges in order of ascending weight
for each edge in sorted order do
    colour edge red if both endpoints are in the same blue forest (forming a cycle)
    otherwise colour blue
end for
return the tree formed by the blue edges
```

To prove that the algorithm is correct, all that remains is to show that these are correct applications of the colouring rules (which follows from the ordering used).

For quickly checking the forests that the endpoints belong to, the union-find data structure can be used:

Algorithm 10.3 Kruskal's algorithm with union-find

```
all edges begin uncoloured
for each v ∈ V do
    MAKE-SET(v)
end for
sort edges in order of ascending weight
for each edge (u, v) in sorted order do
    if FIND(u) ≠ FIND(v) then                    ▷ Joining different trees
        colour (u, v) blue
        UNION(u,v)
    else
        Colour edge red
    end if
end for
return the tree formed by the blue edges
```

A common implementation of union-find uses a collection of trees (one per set). The representative of each set is the root of the tree.

We can use some heuristics to ensure that the trees remain small and that each node has few children:

- Path compression: when calling FIND(x), change the parent pointer of every node encountered on path to x to point to the root
- Weight union: when performing a union operation, the heavier (containing more nodes) tree is the one whose root is used as the root as the new tree
- Height union: like weight union but using height of tree instead of weight

Theorem 10.4. *A sequence of n MAKE-SET operations and m FIND and UNION operations using path compression and one other heuristic has worst-case time complexity $O(n + m\alpha(n))$ where α is the inverse Ackerman function.*

Corollary 10.5. *Kruskal's algorithm runs in $O(n + m \log n)$ time, or $O(n + m\alpha(n))$ for integer weights (as radix sort can be used).*

10.3 Prim's algorithm

Algorithm 10.4 Prim's algorithm

```
Initially, all edges are uncoloured, T is a tree with one arbitrary vertex
for n − 1 times do
    Colour blue a min-weight edge incident to T (connecting to a vertex not in T) and add it to T
end for
return T
```

Using adjacency lists, Prim's algorithm can be made to run in $O(n(n + m))$ time - $n - 1$ iterations times $O(n + m)$ work per iteration to find min-weight adjacent edge.

Define $d(v)$ to be the cost of the lightest edge between T and $v \in V \setminus T$, and $\pi(v)$ to be the other endpoint of this edge. This suggests a faster algorithm, keeping track of these values instead of recalculating them from scratch on each iteration:

Algorithm 10.5 Template algorithm for faster Prim's

```
Let s be an arbitrary vertex in V
T := ∅                                   ▷ No edges, although we still consider s to be in T
Initialise π and d values for vertices adjacent to s
for n − 1 times do
    Let v be a vertex with minimum d(v)
    T := T ∪ {v, π(v)}
    d(v) now is no longer defined (no longer a non-tree vertex)
    Update all other π and d values to take in to account v now being in tree
end for
Return T
```

Of course, the crux of the problem is now how to quickly update d and π once a vertex is added to the tree. General approach:

Algorithm 10.6 Updating values for Prim's

```
for each vertex u ∈ V \ T which is adjacent to v do
    if w(v, u) < d(u) then
        d(u) := w(u, v)
        π(u) = v
    end if
end for
```

As is, this would take $O(n^2)$ time: $O(n)$ for initialisation and finding v that minimises $d(v)$, and then $O(deg(v))$ to update other values; multiplied by $n-1$ iterations. To improve on this, we can use a priority queue. E.g. with a heap this will now be $O(\log n)$ to find the minimising vertex and $O(deg(v) \log(n))$ to update the values (using DECREASE-KEY), for a total of $O(n + m \log(n))$ when summing over all edges. Fibonacci heaps are faster yet: FIND-MIN takes constant time, DELETE-MIN takes $O(\log n)$, and DECREASE-KEY is constant; overall this is $O(n \log(n) + m)$.

10.4 Borůvka and round-robin algorithms

Amenable to parallel implementation.

Algorithm 10.7 Borůvka's algorithm

```
Initialise with all edges ungecoloured
repeat
    for each blue tree T do
        Colour blue an edge of minimum weight adjacent to T
    end for
until there is a blue tree containing all vertices
return the blue tree
```

Algorithm 10.8 Round-robin algorithm

Initialise $Q := V$ ▷ A queue of sets that are connected components by blue edges
for $n - 1$ times **do**
  Let A be first element in queue Q
  Apply blue rule to the partition A and $V \setminus A$
  Let $\{x \in A \in Q, y \in B \in Q\}$ be the blue edge from this applications
  Delete both A and B from Q
  Add $A \cup B$ to Q
end for
return the blue tree

The round-robin algorithm can be seen as a variant of Borůvka's algorithm.
Consider the round-robin algorithm as operating in **stages**; a stage ends once all sets that were in Q at the start of the stage has been deleted.

Lemma 10.6. *Any set added to Q in stage k will have cardinality as least 2^k.*

Proof. By induction: each set is at least a doubling of the bound from the previous stage. □

Corollary 10.7. *There are at most $\log n$ stages.*

In a straightforward implementation, each stage takes $O(m)$ time as we are considering edges that are incident to a partition of the graph (in worst-case, we must consider every edge). So $O(m \log n)$.

Round-robin in planar graphs

Planar graphs are sparse (at most $3n$ edges) and closed under contraction (?), so can achieve linear time by contracting blue trees after each stage (keeping the lightest inter-tree edge).

Algorithm 10.9 Planar round-robin

$k := 1, G_k = G$
while G_k has more than one vertex **do**
  Run a stage of round-robin
  Contract G_k and store result as G_{k+1}
  Increment k
end while

At the end of a stage, the number of vertices is at most half of what it was at the start of the stage. So the number of vertices in G_k is at most $\frac{n}{2^{k-1}}$. Total running time:

$$\sum_{k=1}^{\log n} O(|G_k|) = \sum_{k=1}^{\log n} O(\frac{n}{2^{k-1}}) = 2n \sum_{k=1}^{\log n} O(\frac{1}{2^k}) = 2nO(1 - 2^{\log \frac{1}{n}}) = O(n)$$

10.5 Fast round-robin

With a bit of magic, round-robin can be implemented in $O(m \log \log n)$ time. Idea:

- Run $\log \log n$ stages of round-robin. This will leave us with at most $\frac{n}{\log n}$ sets, as each set has size at least $2^{\log \log n} = \log n$
- Contract the sets that remain to form G', and aim to run each subsequent stage in $O(\frac{m}{\log n})$ time. When doing the contraction, we must keep the lightest edge between each set.
- Modify G': let U be a a vertex in G' (i.e. a set of vertices of G) and divide the edges incident to U into $\lceil \frac{deg(U)}{\log n} \rceil$ lists each of size at most $\log n$. Sort each group of edges by weight (each group has $\log n$ edges so this takes $O(m \log \log n)$ time). Now, call a modified version of round-robin on G' as below.

Modified algorithm:
When we are processing some set $X \in Q$ from the queue of supervertices, we seek the lightest adjacent edge. Iterate for each $U \in X \in Q$. To find the lightest edge adjacent to U that leaves X, iterate through all the sorted lists we formed for U taking an edge that leads out of X (deleting any lighter edges that *do* lead to X that are encountered along the way - these are red). Upon completion of this, we have one such edge per list: $\lceil \frac{deg(U)}{\log n} \rceil$ in total. So we can find the lightest such edge in $O(1 + \frac{deg(U)}{\log n})$ time plus the number of edges we had to discard.
Now, there are $O(\log n)$ stages, so consider the time spent on finding edges adjacent to U over all the stages:

$$\log(n) O(1 + \text{number discarded} + \frac{deg(U)}{\log n})$$

Each edge is deleted at most once, so over all the stages the number discard sum to at most $deg(U)$, and we get:

$$= O(\log n + deg(U))$$

There are at most $\frac{n}{\log n}$ sets so the total run time on G' is

$$O(n + m)$$

So the overall runtime is $O(m \log \log n)$, from the sorting and intial stages.

10.6 Summary

- Kruskal's with UNION-FIND is $O(m \log n)$
- If edges are sorted (or quickly sortable) then this becomes $O(m\alpha(n))$
- Prim's has a simple $O(n^2)$ implementation
- With priority queues, $O(m \log n)$
- With Fibonacci heaps $O(m + n \log n)$
- Borůvka's algorithm is good for parallel implementation
- Round-robin is vertex-linear for planar graphs and has a $O(m \log \log n)$ implementation

11 Matchings

A matching is a subset $H \subset E$ such that no two edges in H share an endpoint. We seek to find to matching of maximum weight or cardinality.

11.1 Characterisation

Theorem 11.1. *A matching M is maximum if and only if it admits no augmenting path: a path that is alternating, and starts and ends at a free (unmatched) vertex.*

So can design algorithms for maximum matching by finding augmenting paths.

11.2 Bipartite matching

Can be formulated as a max-flow problem, which can in turn be solved in $O(nm)$ time (where edges are directed and have capacities and we seek to send 'flow' at a level at most the capacity and the flow in and out of vertices are equal except for the start s and end vertex t).

Algorithm 11.1 Reducing bipartite matching to a max-flow problem

Let $G = (L \cup R, E)$ be the input
Let G' be $(L \cup R \cup \{s, t\}, E')$ with E' directing all $e \in E$ from L to R with infinite capacity and unit capacity edges from s to L and from R to t
return max-flow(G')

Theorem 11.2 (Correctness of previous algorithm)**.** *Proof.* Firstly, the size of a maximum matching in G is at most the max-flow in G: given any matching, it immediately suggests a flow of the same capacity by sending unit flow through each matched edge.

Additionally, the size of a maximum matching in G is at least the max-flow in G: Let $f : E' \to \mathbb{Z}$ be a max-flow in G' (we may assume integrality). We may also assume $f(E') = \{0, 1\}$ (why??). Then we can find a matching by taking all edges that get mapped to 1 under f and hence has the same size as the max flow.

Both $\geq$ and $\leq$ have been shown, proving that the quantities are equal. □

Another algorithm is to grow an existing matching by constructing an alternating tree to find augmenting paths.

There is a straightforward $O(n(n + m))$ algorithm to find an augmenting path: try to find an alternating tree from each free vertex. This then suggests a $O(n^2(n + m))$ to find a maximum matching as there cannot be more than $O(n)$ edges in a matching. There are other approaches that reduce the cost of finding an augmenting path and reduce the number of augmentations required.

The next theorem helps; if we have checked for augmenting paths from a vertex, we don't need to check from there again:

Theorem 11.3. *Let M be a matching in G, u be free WRT M, P be an augmenting path for M. Then, if there is no augmenting path from u in M there is also no such path from u in $M \oplus P$.*

Proof. Assume for contradiction that there does not exist an augmenting path from u WRT M but there does exist an augmenting path P' from u WRT $P \oplus M$ (where P is some augmenting path).

If P' and P are node-disjoint, it is an immediate contradiction, as then P' is also an augmenting path WRT M.
Otherwise, let v be the first node in common and split the paths at v. There is now a way to recombine the paths to get an augmenting path from u WRT M ↯.

□

Algorithm 11.2 Faster bipartite matching: no more than one augmenting tree per vertex

```
The input is a graph G
Set M as an empty dictionary structure, free := |V|
for x ∈ V do
    M[x] = 0
end for
r := 0                                              ▷ Current root for alternating tree
while free ≥ 1 and r < n do
    r := r + 1
    if M[r]=0 then                                  ▷ If the tentative tree root is a free vertex
        for i = 1 to n do
            T[i] = 0                                ▷ Intialise a tree, with T[a] denoting the parent of a
        end for
        Q := ∅, enqueue r to Q, augmenting:=false   ▷ Do BFS with Q as the queue
        while augmenting=false do
            dequeue from Q, storing the dequeued vertex as x
            if there is a vertex adjacent to x s.t. M[x] = 0 then   ▷ Free adjacent vertex
                Augment with respect to the augmenting path found
                augmenting:=true, free := free − 1
            else                                    ▷ Grow the tree
                for every y adjacent to x do
                    if T[y]=0 then                  ▷ Not yet in the tree
                        T[y] := x
                        Enqueue M[y] to Q
                    end if
                end for
            end if
        end while
    end if
end while
```

Hopcroft-Karp can do better yet - in time $O(\sqrt{n}(n+m))$, by finding multiple paths at once, of ever-growing lengths. There cannot be many long disjoint paths as the next lemma states:

Lemma 11.4 (Bound on length of augmenting paths)**.** *Let M^* be maximum matching, and M any matching. If M's shortest augmenting path has length k, then $|M^*| - |M| \leq \frac{|V|}{k}$.*

Proof. Consider the graph with $M \oplus M^*$ as its edge set. It contains at least $|M|-|M^*|$ vertex-disjoint augmenting paths with respect to M (as these contribute one extra edge to M; cycles contribute equally to both), and each have length $\geq k$. And the total length of the paths is $\leq |V|$ (otherwise there would be a cycle), so result follows. □

In each phase of Hopcroft-Karp, construct a maximal set Π of disjoint augmenting paths. Denote by $M \oplus \Pi$ augmentation WRT each path in Π.

Lemma 11.5 (After augmenting by a maximal set of disjoint shortest augmenting paths, the length of a shortest augmenting path increases)**.** *Let k be the length of a shortest augmenting path WRT M and Π a maximal set of disjoint shortest augmenting paths WRT M. Then, the length of the shortest augmenting path WRT $M \oplus \Pi$ is greater than k.*

Proof. Consider P, a shortest augmenting path WRT $M \oplus \Pi$.
If P is disjoint from every vertex in Π's paths, then result clearly holds as otherwise Pi is not maxmimal. So assume it intersects $P_1, \ldots, P_t \in \Pi$ in that order.
Then construct shorter paths R_i, etc... □

Define G_M for a matching M to be the directed graph with unmatched edges pointing from L to R and vice versa. Let L^*, R^* be the set of free vertices in L and R respectively, and $d(v)$ be the distance from L^* to v. Define the *layered graph* G_M^* to contain all edges (u, v) from G_M that have $d(v) = d(u) + 1$ (i.e. the edges that lie on some shortest path between vertices; those found by BFS). Then, all paths between vertices in G_M^* are shortest paths in G_M.
Now, to find a maximal set of shortest vertex-disjoint augmenting paths, run DFS from L^* until reaching R^* for the first time and delete all visited vertices. Repeat, returning each shortest path found.
All of the above can be done in time $O(m + n)$, with BFS/DFS.
We only need $O(\sqrt{n})$ augmentations: each iteration causes the length of the shortest augmenting path to increase by 1, so after $\sqrt{n}$ iterations the length of a shortest augmenting path is $\sqrt{n}$. At that point, at most $\frac{n}{\sqrt{n}} = \sqrt{n}$ augmenting paths remain, by lemma. So augment $2\sqrt{n} = O(\sqrt{n})$ times. Since each augmentation is $O(n + m)$, this yields $O(\sqrt{n}(n + m))$.

Algorithm 11.3 Hopcroft-Karp

$M := \varnothing$
augmenting found=true
while augmenting found is true **do**
  Set Π to be a maximal set of vertex-disjoint shortest augmenting paths (do this using above method of BFS then DFS)
  $M := M \oplus \Pi$
  if $\Pi = \varnothing$ set augmenting found to false
end while
return M

Definition 11.6 (Neighbourhoods of sets of vertices)**.** *For a graph $G = (V, E)$ and $S \subset V$, the neighbourhood $\Gamma(S)$ is defined as*

$$\bigcup_{v \in S} \{u \in V : u \text{ is adjacent to } v\}$$

Theorem 11.7 (Hall's theorem)**.** *A bipartite graph $G = (L \cup V, E)$ has a perfect matching (matches all vertices in L) if and only if it satisfies*

$$\forall S \subseteq L : |\Gamma(S)| \geq |S|$$

Proof. By induction: pick $u \in L$ and $v \in R$, match them and remove. If the smaller graph satisfies Hall's condition, we are done.

Else, we are in the case where there exists a a set $S \subseteq L \setminus \{u\}$ such that $|\Gamma(S) \setminus \{v\}| < |S|$. Apply magic by letting S be the smallest such set.

We have that $|\Gamma(S)| \geq |S|$, so S has exactly $|S|$ neighbours in R. And by minimality, the subgraph induced by $S \cup \Gamma(S)$ satisfies Hall's condition! So there is a perfect matching of vertices in S to $\Gamma(S)$.

And there is also a matching of vertices in $L \setminus S$ to vertices in $R \setminus \Gamma(S)$: consider $A \subset L \setminus S$. $|\Gamma(A \cup S)| \geq |A \cup S| = |A| + |S| = |A| + |\Gamma(S)|$. This then implies that A alone has at least $|A|$ neighbours in $R \setminus \Gamma(S)$ as S could only have contributed $|S|$ of the above quantity.

□

Corollary 11.8. *Every d-regular bipartite graph contains d edge-disjoint perfect matchings*

11.3 Vertex covers and matchings: König's theorem

Definition 11.9. *A vertex cover is a set of vertices such that every edge has an endpoint in the set.*

Lemma 11.10 (Weak duality)**.** *Let M be a matching and C a vertex cover. Then $|M| \leq |C|$.*

Proof. Consider the function $M \to C$ that maps an edge $e \in M$ to the vertex in C that contains one of its endpoints. By the definition of a matching this function is injective, as all edges in M are vertex-disjoint. □

Lemma 11.11 (Strong duality for bipartite graphs)**.** *Given a matching M in a bipartite graph, a vertex cover of size $|M|$ can be constructed.*

Proof. Define a directed graph G_M that directs unmatched edges from L to R and vice versa. Let Q be the set of all vertices reachable from free vertices in L using edges in G_M. Let $C = (L \setminus Q) \cup (R \cap Q)$.

Then C is a vertex cover and $|C| = |M|$:

Firstly, note that all free vertices in L are in Q. And no free vertices in R can be in Q or else we would have an augmenting path.

Assume that C is not a vertex cover. Then there exists an edge $e = \{x, y\}$ with $x \in L \cap Q$ and $y \in R \setminus Q$. e cannot be matched; if it was then e would point from R to L and we would need $y \in Q$ for x to be reachable from a free vertex in L. So e is unmatched and is directed from L to R. But then y becomes reachable using e! In either case a contradiction has been reached, so C is a vertex cover.

By the previous lemma, it now suffices to show $|C| \leq |M|$. Note that every vertex in C is matched: all vertices in $L \setminus Q$ are matched by definition, and if there is a free vertex in $R \cap Q$ it would form an augmenting path. There cannot be a matched edge from $R \cap Q$ to $L \setminus Q$ (otherwise the endpoint would be reachable) so the edges that match all vertices in C are distinct. So $|C| \leq |M|$, implying $|C| = |M|$ by weak duality. □

Corollary 11.12 (König's theorem)**.** *In a bipartite graph, the size of a maximum matching is the same as the size of a minimum cover.*

11.4 Maximum matching in general graphs

In general (not necessarily bipartite) graphs, a similar approach of finding alternating trees can be applied. But now there is a new case that can occur - encountering an edge between layers of the same parity. Whenever this occurs, there encountered edges would form an odd-length cycle in the alternating tree. Such cycles are called **blossom**s. (This never occurs in bipartite graphs due to the lack of odd cycles). The **stem** of a blossom is the edge closest to the root of the alternating tree that is incident on the blossom.

To handle this case, we "take quotients" with respect to each blossom as soon as it is encountered - i.e. we form a new graph G' and a new alternating tree T' that considers the blossom as a single vertex.

Then, once an augmenting path in the final alternating tree has been found, an augmenting path in the original graph can be found by following a path around the blossom. The correctness of this algorithm follows from a lemma:

Lemma 11.13. *There is an augmenting path in G' if and only if there is an alternating path in G*

Proof. For the forward direction, assume that there is an augmenting path in G'. If this augmenting path is vertex-disjoint from the blossom we are done. Otherwise, consider the unmatched edge e in the augmenting path that is incident on the blossom. There is an edge in the blossom that it matched and is incident to e. Then, following this edge back to the stem of the blososm and add the resulting path to the augmenting path to obtain an augmenting path in G. (The case of the augmenting path starting at the blossom is omitted for brevity.)

Conversely, assume that there is an alternating path in G, and then a blossom is "shrunk" (replaced by a single vertex). The edge at the stem of the blossom must be matched (due to the parity) $\square$