Graph Algorithms

Rune Lyngsø

October 26, 2009

1 Graphs

Despite its simplicity, graphs are a very versatile concept with a myriad of applications. It captures the notion of entities (nodes) and their interactions (edges). Standard examples of graphs in everyday life include road networks, with intersections the nodes and an edge between intersections if they are connected by a stretch of road; social networks, with people the nodes and edges connecting people that are friends or know each other; resource constraints, e.g. what teachers and what rooms that can be used to teach which subjects; task prerequisites, e.g. that certain courses need to be taken before enrolling on another course, or that the outer walls of a house need to be raised before work on the roof can begin. Also in biology do we see a fair number of systems that can naturally be described using graphs. Fig. 1 gives just a few examples.

Graphs come in different flavours, depending on the system they are used to represent. This ranges from ordinary undirected graphs to weighted directed hypergraphs. In addition, multigraphs further generalise the graph concept by allowing multiple edges to connect the same two nodes – one example being the two intersections where a road with Crescent as its surname joins the main road. Below we give definitions of types of graphs, ignoring the multigraph generalisation.

Definition 1 (Undirected Graph)**.** An undirected graph $G = (V, E)$ is a pair of a set V of nodes and a set $E \subseteq \{\{u, v\} \mid u, v \in V \wedge u \neq v\}$ of undirected edges (unordered pairs of nodes).

Definition 2 (Directed Graph)**.** A directed graph $G = (V, E)$ is a pair of a set V of nodes and a set $E \subseteq \{(u, v) \mid u, v \in V \wedge u \neq v\}$ of directed edges (ordered pairs of nodes).

Figure 1: This spot will eventually contain some example networks: A signalling pathway, an alternative splicing graph, and ancestral recombination graph and a KEGG metabolic network

In some contexts graphs can be mixtures between undirected and directed graphs, for example if edges have been established by a mixture of experiments where some reveal causation while other only reveal dependence. Usually these can be viewed as directed graphs with each undirected edge $\{u, v\}$ replaced by the two directed edges (u, v) and (v, u). There can be issues if an undirected edge should be viewed as a directed edge with unknown direction, though.

Definition 3 (Undirected Hypergraph)**.** An undirected hypergraph $G = (V, E)$ is a pair of a set V of nodes and a set $E \subseteq 2^V \setminus \emptyset$ of hyperedges (non-empty subsets of nodes).

Here 2^V is used to denote the power set of V, i.e. all subsets of V.

Definition 4 (Directed Hypergraph)**.** A directed hypergraph $G = (V, E)$ is a pair of a set V of nodes and a set $E \subseteq \{(U, V) \mid U, V \subseteq 2^V \setminus \emptyset\}$ of directed hyperedges (pairs of non-empty subsets of nodes).

For a directed hyperedge $e = (U, V)$ we will usually denote U the in-set of e and V the out-set of e. Depending on context, it may be required that $U \cap V = \emptyset$; the requirement that both U and V are non-empty may also be relaxed, though usually at least one will be required to be non-empty.

Definition 5 (Weighted Graph)**.** A weighted, (un)directed (hyper)graph $G = (V, E, w)$ is a triplet where $G' = (V, E)$ is a (un)directed (hyper)graph and $w : E \mapsto \Omega$ assigns a weight from Ω to each edge.

In these notes it will generally be assumed that $\Omega = \mathbb{R}$, and for most of the problems considered here Ω is required at least to be a totally ordered set with an additive operation.

2 Graph Traversal

A fundamental technique for graphs is the ability to traverse them, i.e. visit all nodes and edges in the graph, in a certain order. Exactly how this is done does depend on the representation chosen, which again may depend on implementation language choice. We will assume that we have some way of iterating through all nodes in a graph, and for each node iterating through all edges. We will also assume that we have some way of efficiently marking a node as already visited and check whether a node has been thus marked. If nodes are $V = \{1, \ldots, n\}$ this can be done with a list; if nodes are represented by a complex data structure the mark may be added as part of this data structure. The main aim of this section is to define some basic elements of graph traversal that will be used, possibly in modified form, in ensuing sections.

Possibly the simplest way to traverse a graph in an ordered fashion is, for each node we visit to visit all nodes it is connected to (and recursively the nodes it is connected to)

Algorithm 1 Depth first search

Input: Graph $G = (V, E)$
  define depth-first-search(u)**:**
    if u is not visited **then**
      Mark u as visited
      for $(u, v) \in E$ **do**
        depth-first-search(v)
  for $u \in V$
    depth-first-search(u)

in turn. This is known as depth-first search and illustrated in Alg. 1. The depth-first refers to the algorithm always trying to extend the current path, moving further and further away from the starting node; only when no more unvisited nodes can be reached from the current node does the algorithm backtrack and continue from nodes closer to the starting node. Assuming that each new node and each new edge can be obtained in constant time, the algorithm obviously runs in time O $(|V| + |E|)$ (order of $|V| + |E|$, i.e. the time grows no faster than a function proportional to $|V| + |E|$) as each edge is considered only once (twice for undirected graphs) and nodes only once apart from visits attributable to incoming edges. Evidently each node and each edge is visited by the algorithm. As it stands, it just traverses nodes and edges, but it can easily be modified, e.g. to compute the number of different transcripts that can be generated from an alternative splicing graph.

Algorithm 2 Generic search from node

Input: Graph $G = (V, E)$, node s, and structure L allowing insertion and extraction
  Insert s in L
  while L is not empty **do**
    Extract node u from L
    if u is not visited **then**
      Mark u as visited
      for $(u, v) \in E$ **do**
        Insert v in L

Depth-first search can be seen as one variant of a generic search strategy: whenever we encounter an unvisited node mark it as visited and set up all nodes connected to it for a future visit. This strategy is formalised in Alg. 2. If L is a stack, i.e. the first element to be extracted is the last inserted, we get exactly the behaviour of the depth-first search algorithm – whenever a node is visited, the next node to be visited will be an unvisited neighbour if any such exists.

By using a queue structure for L, i.e. the first element to be extracted is the first

inserted, Alg. 2 results in a breadth-first search strategy: the first nodes visited will be those directly connected to the start node u, then the nodes reachable by traversing two edges will be visited, and so on. This is very useful in settings where proximity to a particular node is of interest, for example when we are concerned with the effects of a particular gene in a regulatory network – the more intermediates we need to go through, the lesser the expected effect will be. When V is a set of configurations a system can be in and two configurations are connected if we can change one into the other in a single step, breadth-first search will result in most parsimonious series of steps converting starting configuration s into any other configuration.

List and queues can be implemented such that both insertion and extraction can be done in constant time. Hence, we can use Alg. 2 to implement both depth-first and breadth-first search in time $\mathrm{O}(|V| + |E|)$. In the following section we will see how one of the standard shortest path algorithms for weighted graphs is obtained by using a priority queue structure for L.

3 Single Source Shortest Path

Breadth-first search is sufficient for determining the minimum distance from a node u to all other nodes in the graph when all edges are born equal. For example a person's Erdös number is the distance from that person to Paul Erdös in the graph where two people are connected by an edge if they have coauthored a paper. In a high throughput protein interaction experiment we would also usually only measure whether two proteins bind together but not the strength with which they bind. However, in a lot of cases edges will carry a weight. A classic example would be road networks where there would be a distance (or time or cost) associated with each stretch of road connecting two intersections. In regulatory networks we could have a measured regulatory effect, in metabolic networks energy use for each reaction, and in alternative splicing graphs the probability of choosing a particular acceptor site for a given donor site. For a weighted graph the single source shortest path problem is to compute the minimum path length from a given node u to all other nodes in the graph. The following definition formalises the notion of paths being a series of nodes connected by edges, the length of a path being the sum of edge lengths, and a shortest path being exactly that.

Definition 6 (Path, length of path, and shortest path)**.** Let $G = (V, E, w)$ be a (directed or undirected) weighted graph. A path π from s to t, $s, t \in V$, is a sequence of nodes $s = u_0 \to u_1 \to \ldots \to u_k = t$ such that $\forall 1 \leq i \leq k : (u_{i-1}, u_i) \in E$. The length of a path $\pi = u_0 \to \ldots \to u_k$ is $w(\pi) = \sum_{i=1}^{k} w(u_{i-1}, u_i)$. A path $\pi = s \to \ldots \to t$ is a shortest path from s to t iff $w(\pi) = \min \{w(\pi') \mid \pi' \text{ is path from } s \text{ to } t\}$.

3.1 Dijkstra's Algorithm

Let $G = (V, E, w)$ be a (directed or undirected) weighted graph and $s \in V$ the node we want to compute the distance from. If we let $d[u]$ denote the distance, i.e. length of shortest path, from s to u, then we need to solve the set of equalities given by

$$d[u] = \min_{(v,u)\in E} \{d[v] + w(v, u)\} \tag{1}$$

for all $u \in V \setminus \{s\}$ and $d[s] = 0$. This follows from the obvious observation that if $\pi = s = v_0 \to v_1 \to \ldots \to v_k = u$ is a shortest path from s to u, then any prefix $s = v_0 \to \ldots v_i$ will be a shortest path from s to v_i. If not, we could replace the $s = v_0 \to \ldots v_i$ part of π with a shorter path from s to v_i to obtain a shorter path to u.

Dijkstra's algorithm is only guaranteed to work if all edge weights are non-negative, i.e. $\forall e \in E : w(e) \geq 0$. For road networks this is a quite natural assumption, but e.g. metabolic networks may have some reactions that are energy producing rather than energy consuming. In the remainder of this section we will assume that all edge weights are non-negative, and return to the problem with negative edge weights in the next section.

Consider the neighbours of s in G. If u is a neighbour with $w(s, u)$ minimal for all edges from s, then $d[u] = w(s, u)$. Otherwise there would be a path $\pi = s \to v_1 \to \ldots \to v_k = u$ with $w(\pi) = w(s, v_1) + \sum_{i=2}^{k} w(v_{i-1}, v_i) < w(s, u) \Leftrightarrow \sum_{i=2}^{k} w(v_{i-1}, v_i) < w(s, u) - w(s, v_1) \leq 0$. Consequently at least one edge in π would need to have a negative weight.

If v is another neighbour of s with non-minimal edge weight, we cannot be sure that $d[v] = w(s, v)$. For example, we could have $(u, v) \in E$ with $w(s, u) + w(u, v) < w(s, v)$. However, we can build on the idea of repeatedly choosing the best candidate we do not yet have the distance for. Assume that $S \subseteq V$ with $s \in S$ has the property that $\forall u \in S, v \in V \setminus S : d[u] \leq d[v]$, i.e. S is a set of $|S|$ nodes closest to s. Let $d'[v] = \min_{u\in S} \{d[u] + w(u, v)\}$ for all $v \in V \setminus S$. If v is a best candidate among the remaining nodes, i.e. $d'[v] = \min_{u\in V\setminus S}\{d'[u]\}$, then $d[v] = d'[v]$ and $\forall u \in V \setminus S : d[u] \geq d[v]$.

To see that $\forall u \in V \setminus S : d[u] \geq d[v]$, consider a shortest path $\pi = s \to u_1 \to \ldots \to u_k$ to a node $u \in V \setminus S$. This must start in S as $s \in S$. Let u_i be the first node on the path not in S. Then $d[u] = w(\pi) = d[u_{i-1}] + w(u_{i-1}, i) + \sum_{j=i+1}^{k} w(u_{j-1}, u_j) = d'[u_i] + \sum_{j=i+1}^{k} w(u_{j-1}, u_j) \geq d'[v] + \sum_{j=i+1}^{k} w(u_{j-1}, u_j)$. It now follows from (1) that

$$\begin{aligned} d[v] &= \min_{(u,v)\in E}\{d[u] + w(u, v)\} \\ &= \min \left\{ \min_{(u,v)\in E, u\in S}\{d[u] + w(u, v)\}, \min_{(u,v)\in E, u\notin S}\{d[u] + w(u, v)\} \right\} \\ &= d'[v] \end{aligned}$$

as $d'[v] = \min_{(u,v)\in E, u\in S}\{d[u] + w(u,v)\}$ and $\min_{(u,v)\in E, u\notin S}\{d[u] + w(u,v)\} \geq \min_{(u,v)\in E, u\notin S}\{d'[v] + w(u,v)\}$.

So to compute shortest distances from s to all other nodes in G we just need to apply Alg. 2 with a data structure L that allows us to insert nodes with a priority (the value of $d'[v] = d[u] + w(u,v)$ in the last line of the algorithm and 0 when we insert s at the beginning) and extract the remaining element with lowest priority. Such data structures are called *priority queues* and usually also allow updates of priorities, such that we can just update the priority of v in the last line of Alg. 2 rather than having multiple occurrences of it in L. It is worth mentioning that this update step is commonly known as *relaxation*, as it relaxes the constraint that $d[v] \leq d[u] + w(u,v)$.

Priority queues can be implemented in a variety of ways. For example, we can just maintain $d'[v]$ for all remaining nodes and whenever we need to extract a node from L run through all remaining nodes and choose one with a minimum value. This requires $\mathrm{O}(1)$ (constant) time for each relaxation and $\mathrm{O}(|V|)$ time for each extract. As we are extracting each node once and relax each edge once, the total time required would be $\mathrm{O}\left(|V|^2 + |E|\right)$. For dense graphs $|E| \propto |V|^2$. As we cannot in general solve the shortest path problem without considering each edge at least once, this simple approach is optimal for dense graphs. We could also keep a sorted list of d' values for remaining nodes. This would allow extracting nodes in time $\mathrm{O}(1)$ but on the other hand increase the time required for relaxation to $\mathrm{O}(|V|)$, resulting in total time required of $\mathrm{O}(|V| + |E||V|)$ – usually $|E| \geq |V|$, so of these simple solutions the first is preferable.

More complicated structures can be used for the priority queue for non-dense graphs, if further speed-up is necessary. Without going in to details of how it works, *Fibonacci heaps* allow the next node to be extracted in time $\mathrm{O}(\log|V|)$ and relaxation of an edge in time $\mathrm{O}(1)$, resulting in total time required of $\mathrm{O}(|V|\log|V| + |E|)$.

3.2 Bellman-Ford Algorithm

With negative edge weights, there is a possibility for cycles of negative weight in a graph. That is, a cyclic pathπ $u_0 \to u_1 \to \ldots \to u_k = u_0$ with $w(\pi) < 0$. When a graph contains one or more negative weight cycles that can be reached from s, the shortest path distance is no longer well defined. Any node on such a cycle or reachable from a node on a such a cycle can be reached from s on a path with an arbitrarily small weight by just adding more and more trips around the negative weight cycle. If no negative weight cycles exist in G, the shortest path distance is well defined, but may not be computable by Dijkstra's algorithm.

We can however still use the relaxation technique. If there are no negative weight cycles in G, there is a shortest path from s to any node v traversing at most $|V| - 1$ edges – there can be no positive weight cycles on a shortest path, and zero weight cycles can be excised with no effect on the path length. By the argument in the previous section that

any prefix of a shortest length path is itself a shortest length path, we furthermore have that if any shortest path to node u needs to traverse at least k edges, there is a node v with a shortest path traversing only $k-1$ edges and where a shortest path to u has v as immediate predecessor to u. So we can iteratively compute shortest path distances where paths are restricted to traverse only one edge, then two edges, etc. When distances have been computed allowing the traversal of $|V|-1$ edges we should have shortest distances from s to all other nodes. If any of the inequalities $d[u] \leq d[v] + w(v,u)$ implicit in (1) remain unsatisfied, the graph contains one or more negative weight cycles.

Algorithm 3 Bellman-Ford Single Source Shortest Path Algorithm

Input: Weighted graph $G = (V, E, w)$ and source node s
  Set $d[s] = 0$, $d[u] = \infty$ for $u \in V \setminus \{s\}$
  for $i = 1$ **to** $|V| - 1$ **do**
    for $(u, v) \in E$ **do**
      $d[v] = \min\{d[v], d[u] + w(u, v)\}$ (Relax edge (u, v))

This is known as the Bellman-Ford algorithm, and formalised in Alg. 3. For each outer iteration we run through all edges and perform a relaxation step that can be done in time $\mathrm{O}(1)$. Hence, the total time for running the Bellman-Ford algorithm is $\mathrm{O}(|V||E|)$. By terminating as soon as no distances where updated in the inner iteration, this can be modified to $\mathrm{O}(\ell|E|)$ where ℓ is the maximum over all nodes of the minimum number of edges that needs to be traversed to obtain a shortest path to the node. In the worst case $\ell = |V| - 1$.

3.3 Biasing Search Towards Single Target

So far we have been considering the problem of finding the shortest path distance from a single source to all other nodes in the graph. In many cases, we may only be interested in the distance to one particular node t. Again, road networks is an obvious example – we usually just want to travel from s to t and not from s to all other intersections in the world. Searching for parsimonious solutions in a graph representing the steps that can be taken is another example, for example we may want to find the cheapest explanation that generated an observed set of sequences from a single common ancestral sequence. In general, there is no known way of computing the shortest path distance from a single source to a single destination that is guaranteed to be faster than just computing the shortest path distance to all other nodes. However, this doesn't mean that there is nothing that can be done that won't work in practice – imagine the performance of satellite navigation units and internet driving direction services if they always had to compute distances to all possible destinations.

One simple step that can be taken is of course to terminate computation as soon as we

know the shortest path to t. This point may be difficult to identify in the Bellman-Ford algorithm, but in Dijkstra's algorithm we could terminate when t is extracted from the priority queue. However, if t is far away from s this would not naturally happen until we were almost done anyway.

In Dijkstra's algorithm, we are essentially throwing the net farther and farther out, finding distances to more and more distant nodes (remember the set S in the argument leading to Dijkstra's algorithm). If we could somehow tweak the weights such that edges seemingly leading in the right direction become shorter, we could hope to encounter t at an earlier time in the computation. However, we need to make sure that the tweaking does not lead to incorrect shortest paths.

Assume that for every node $u \in V$ we have a lower bound $\Phi(u)$ on the shortest path distance from u to t. For road networks this could be the direct flight distance from u to t. Now define new edge weights as $w'(u, v) = w(u, v) - \Phi(u) + \Phi(v)$. Consider a path $\pi \equiv s = u_0 \to u_1 \to \ldots \to u_k = u$ from s to some node u. According to the modified edge weights this will have length $w'(\pi) = \sum_{i=1}^{k} w'(u_{i-1}, u_i) = \sum_{i=1}^{k} w(u_{i-1}, u_i) - \Phi(u_{i-1}) + \Phi(u_i) = w(\pi) - \Phi(s) + \Phi(u)$, as the Φ values of all intermediate nodes on the path cancel out. Consequently, the paths from s to u will maintain their relative ordering, all having lengths increased by $\Phi(u) - \Phi(s)$, under the modified weights w'.

The success of this approach of course depends on the quality of the lower bound Φ. In the extreme case where $\Phi(u)$ is the shortest path distance from u to t, an edge (u, v) would have modified weight $w'(u, v) = 0$ iff v would be a possible choice for the first intermediate node on a shortest path from u to t. Hence, performing a depth-first search from s only following edges with modified weight 0 would yield a shortest path from s to t. One other thing to keep in mind when using this approach is that the modified edge weights need to be positive for Dijkstra's algorithm to remain applicable.

4 All Pairs Shortest Paths

Once we can compute the shortest path distance from one source to all other nodes, we can of course repeat the procedure with each node as source to obtain shortest path distances between all pairs of nodes. When edge weights are non-negative and Dijkstra's algorithm can be used, this works well and allow all pairs distances to be computed in time $\mathrm{O}\left(|V|^2 \log |V| + |V||E|\right)$. With negative edge weight we will have to use the Bellman-Ford algorithm, resulting in time $\mathrm{O}\left(|V|^2|E|\right)$ – usually we can do better than this.

In the single source shortest path algorithms we combined paths with edges to create new paths. When maintaining paths between all pairs of nodes we may as well combine paths with paths to create new paths. A simple way to do this would be in a matrix multiplication like fashion. If $D^k_{u,v}$ is the shortest path distance for paths with at most k

edges, shortest path distances allowing twice that number of edges can be computed by

$$D^{2k}_{u,v} = \min_x \left\{ D^k_{u,x} + D^k_{x,v} \right\} \ . \tag{2}$$

Assuming no negative weight cycles, shortest paths contain at most $|V| - 1$ edges, so we only need to apply this procedure $\mathrm{O}(\log |V|)$ times for total time $\mathrm{O}\left(|V|^3 \log |V|\right)$.

Algorithm 4 Floyd-Warshall Algorithm

Input: Weighted graph $G = (V, E, w)$

Initialise $D_{u,v} = \begin{cases} 0 & \text{if } u = v \\ w(u,v) & \text{if } (u,v) \in E \\ \infty & \text{otherwise} \end{cases}$

for $u \in V$ **do**

  for $v, v' \in V$ **do**

    $D_{v,v'} = \min \left\{ D_{v,v'}, D_{v,u} + D_{u,v'} \right\}$

By ordering the nodes and only combining paths at each node in turn, we can cut this to $\mathrm{O}\left(|V|^3\right)$ as illustrated in Alg. 4. For any shortest path π connecting two nodes v, v', there will be a node u that is maximal (i.e. visited last) in the order in which we are visiting the nodes in the outer **for** loop of Alg. 4. Assuming there are no negative weight cycles, we can assume there are no cycles on π so u occurs only one. By the facts that part of a shortest path is also a shortest path, there must be shortest paths from v to u (a prefix of π) and from u to v' (the corresponding suffix of π) that only visit nodes prior to u in the **for** loop. Hence, by a standard induction argument where we assume that distances are correctly computed for all pairs having a shortest path only having intermediate nodes prior to u in the **for** loop order, it follows that the shortest path distance is correctly computed for the v, v' pair.

At the beginning of this section, we stated that with negative edge weights we will have to use the Bellman-Ford algorithm, rather than Dijkstra's algorithm, if we compute all pairs distances by computing distances from each node in turn. While this is true, by just a single application of the Bellman-Ford algorithm, we can get a modified set of edge weights such that original shortest path distances are easily computable from shortest path distances using the modified weights.

We will be using the same technique that was used to bias the single source search towards a single target, defining a function $\Phi : V \mapsto \mathbb{R}$ and setting the modified weight of edge (u, v) to $w'(u, v) = w(u, v) + \Phi(u) - \Phi(v)$. We have already seen how shortest paths remain shortest paths, and that if $d_w[u, v]$ denotes the shortest distance from u to v under edge weights w then $d_w[u, v] = d_{w'}[u, v] - \Phi(u) + \Phi(v)$. All we need is to find a Φ such that w' is non-negative. Define

$$\Phi(u) = \min \left\{ w(\pi) \mid \pi \text{ is a path ending in } u \right\} \ . \tag{3}$$

A minimum weight path ending in u is either empty, or continues a minimum weight path ending at a predecessor v. Hence,

$$\Phi(u) = \min \begin{cases} 0 \\ \min_{(v,u)\in E}\{\Phi(v) + w(v,u)\} \end{cases} \tag{4}$$

It follows that $\Phi(u) \leq \Phi(v) + w(v,u) \Leftrightarrow 0 \leq w(v,u) + \Phi(v) - \Phi(u) = w'(v,u)$ for all $(v,u) \in E$.

To compute Φ we need the one application of the Bellman-Ford algorithm on a slightly extended graph. An extra node s is added to V with edges to all other nodes, i.e. (s,u) will be an edge in the extended graph for every original node u; the new edges are all given weight 0. The shortest path distance from s to u is exactly $\Phi(u)$, as we can start with an edge with weight 0 going to a start node of a minimum weight path ending in u. Evidently the graph extension cannot introduce new negative weight cycles: as there are no edges leading into s, none of the new edges can be part of a cycle. In total, we need to apply the Bellman-Ford algorithm once, modify weights, run Dijkstra's algorithm with each node as source, and finally compute original shortest path distances from shortest path distances computed using the modified weights. This is known as Johnson's algorithm, and requires time $\mathrm{O}\left(|V|^2 \log |V| + |V||E|\right)$, which is in all cases at least as efficient as Alg. 4.

5 Minimum Spanning Trees

Minimum shortest paths give a local view on how best to connect nodes, when we are just interested in connecting pairs of nodes. If we are looking to globally connect all nodes in a graph $G = (V, E, w)$ in the best possibly way, we need a subset $T \subseteq E$ of edges that allow us to get from any node u to any other node v. Moreover, $\sum_{e\in T} w(e)$ should be minimum for all possible subset of edges connecting all nodes. Assuming all edges have positive weights, it is easy to see that any such subset will be a tree, i.e. a graph with no cycles – if there was a cycle in T we could delete one of the edges in the cycle without breaking connectivity, thus lowering the total cost. The problem of finding a minimum cost set of edges spanning the entire graph is known as the Minimum Spanning Tree problem.

The solution to the problem is based on the concept of *light edges*, that allows iterative procedures to build minimum spanning trees by adding one edge at a time. Let $\emptyset \subsetneq B \subsetneq V$ define a bipartition of the nodes. We will call an edge $\{u,v\}$ with $u \in B$ and $v \in V \setminus B$ a light edge for B if it is an edge of minimum weight connecting a node in B with a node in $V \setminus B$, i.e. if

$$w(\{u,v\}) = \min \{w(\{x,y\}) \mid \{x,y\} \in E \text{ and } x \in B, y \in V \setminus B\} \ . \tag{5}$$

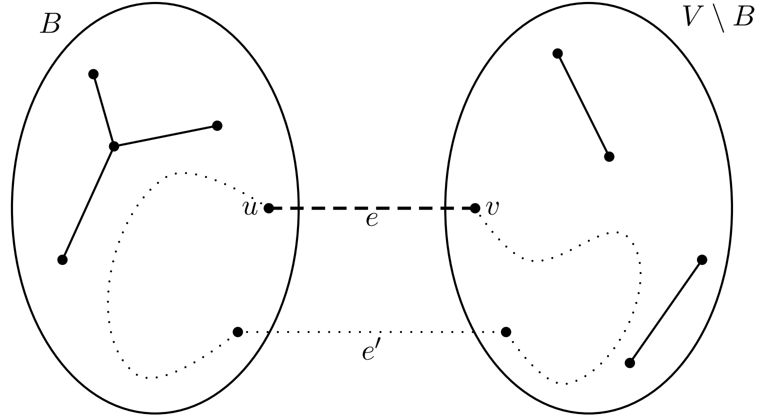

Figure 2: The light edge argument. The solid lines represent the edges in A, B and $V \setminus B$ is a partition with no edges in A connecting the two, the dashed line is the light edge e, and the dotted curve is the path from u to v in T utilising edge e' to cross from B to $V \setminus B$.

Now assume that A is a proper subset of the edges in some minimum spanning tree T. Let B be any bipartition such that there is no edge in A connected a node in B with a node in $V \setminus B$. Finally, let $e = \{u, v\}$ be a light edge for B. We claim that $A \cup \{e\}$ is also a subset of a minimum spanning tree T'.

First, if $e \in T$ then $A \cup \{e\} \subseteq T$. Assume that $e \notin T$. As T is a spanning tree, there will a path connecting nodes u and v. Furthermore, this path with contain at least one edge e' connecting a node from B with a node from $V \setminus B$. If we remove e' from T, this will split T into two disconnected trees. One of these trees will contain u while the other will contain v – we have broken one path connecting u with v and in trees there is only one path connecting two nodes. Hence, adding e will reconnect the two trees T was split into. It follows that $\{e\} \cup T \setminus \{e'\}$ is a spanning tree. As $w(\{e\} \cup T \setminus \{e'\}) = w(T) + w(e) - w(e) \leq w(T)$ due to the fact that e was chosen to have minimum weight over a set also containing e', it must also be a minimum spanning tree.

Using this result, we can grow a spanning tree for G by iteratively adding light edges until we have a tree. One way to do this would be to run through all edges in order of increasing weight and adding an edge to a forest (i.e. set of trees) if it connects two nodes not yet in the same tree. This would require sorting the edges, which is unnecessarily expensive.

Instead we can proceed in a manner very similar to Dijkstra's algorithm, as outlined in Alg. 5. In Dijkstra's algorithm we repeatedly add the next nearest node, were distance is to the source node s we are computing shortest path distances from. In Prim's algorithm we also repeatedly add the next nearest node, but here the distance is defined as

Algorithm 5 Prim's Algorithm

Input: Weighted undirected graph $G = (V, E, w)$
  Insert arbitrary start node $s \in V$ in priority queue Q with key 0 and set $p[s] = \text{None}$
  while Q is not empty **do**
    Extract minimum value node u from Q
    for $\{u, v\} \in E$ **do**
      if $w(\{u, v\}) <$ value v has in Q **and** v not yet in tree **then**
        Set $p[v] = u$ and decrease value of v in Q to $w(\{u, v\})$

the minimum weight of an edge connecting the node to the set of nodes already added. As for Dijkstra's algorithm, we need to extract each node from the priority queue exactly once, and update the value of a node in the priority queue at most once for every edge. Hence the time required by Prim's algorithm to build a minimum spanning tree is $\mathrm{O}(|V| \log |V| + |E|)$. The set of edges in the spanning tree is $\{\{u, p[u]\} \mid u \in V \setminus s\}$, i.e. the $p[u]$ values updated in Alg. 5 tabulates the parent for all nodes except the start node which becomes the root of the tree.

6 Strongly Connected Components

Finding connected components in an undirected graph can be done using either depth-first or breadth-first search (or any other traversal) from each yet unvisited node in turn. For a directed graph $G = (V, E)$, the connectivity problem is a bit more ambiguous, but in many cases it is of interest to identify the strongly connected components. Two nodes $u, v \in V$ are said to be strongly connected if there is a path from u to v, and a path from v to u. Clearly the strongly connected relationship is transitive: ff u and v are strongly connected and v and w are strongly connected, there is a path from u to v and a path from v to w, so there is a path from u to w; there is also a path from w to v and a path from v to um so there is also a path from w to u. Even more obviously, the strongly connected relationship is also reflexive and symmetric. It follows that it defines a partition of V into equivalence classes.

Definition 7 (Strongly Connected Components)**.** The strongly connected components of a graph $G = (V, E)$ is a partition of V into subsets $\{C_i\}_{i=1}^{k}$ such that two nodes $u, v \in C_i$ for some $1 \leq i \leq k$ if and only if u and v are strongly connected.

Identifying the strongly connected components of a graph will often allow us to deal with each component in isolation. For example, in a regulatory network, the expression of two genes only mutually affect each other if they are in the same strongly connected component; in a metabolic network concentrations are only mutually dependent for compounds in the same strongly connected component. More generally, equation systems can

be solved component by component by identifying the strongly connected components in the graph of variable dependencies.

Rather than first describing the intuition behind and then presenting the algorithm, for our strongly connected component algorithm we will first present the algorithm and then explain why it works. It could be argued that this is because there is no intuition to the algorithm; at first sight it does appear to be of little relevance to the strongly connected component problem, consisting of just two depth-first search traversals.

Algorithm 6 Strongly Connected Components

Input: Graph $G = (V, E)$
  Set global variable $t = 0$
  define finishing-time-DFS(u)**:**
    if u is not visited **then**
      Mark u as visited
      for $(u, v) \in E$ **do**
        finishing-time-DFS(v)
      $f[u] = t, t = t + 1$
  for $u \in V$
    finishing-time-DFS(u)
  define reverse-edge-DFS(u, C)**:**
    if u is not visited **then**
      Mark u as visited and add u to component C
      for $(v, u) \in E$ **do**
        reverse-edge-DFS(v, C)
  Initialise set of components $\mathcal{C} = \emptyset$
  for $u \in V$ in order of decreasing value of $f[u]$
    if u is not visited **then**
      Initialise new component C and add C to $\mathcal{C}$
      reverse-edge-DFS(u, C)

In the first depth-first search we record the finishing time for each node, i.e. the order in which we leave the function call where a node was marked as visited. In the second depth-first search we follow edges in the reverse direction, so we find all the nodes from which we can reach a node u. The claim is that all such nodes that have not yet been visited are exactly the nodes in the same strongly connected component as u.

The key observation is that if there is a path from u to v but no path from v to u, then $f[v] < f[u]$. This follows as if we haven't already visited v by the time we commence the visit to u, then we will visit v in a recursive call resulting from the visit to u.

Consider a strongly connected component C of G, and let u be the node in C with maximal finishing time $f[u]$. u will not be visited in any of the reverse edge depth-first

calls for v with $f[v] > f[u]$. If it was, v would be reachable from u. But u must then also be reachable from v as we would otherwise have $f[v] < f[u]$ by the observation above. It follows that we start a reverse edge depth-first search from u in the final **for** loop of Alg. 6.

When we reach u in this **for** loop, any node v from which we can reach u must already have been visited, unless $v \in C$. With a path from v to u there can be no path from u to v – as otherwise $v \in C$ by definition – so $f[v] > f[u]$ and v would have been visited no later than when it was encountered in the **for** loop. Hence, the reverse edge depth-first search initiated from u will not visit any node not in C.

Conversely, when we reach u in the **for** loop, no node $v \in C$ can have been already visited. Otherwise there would be a node w that could be reached from v with $f[w] > f[u]$. But v can be reached from u so w can be reached from u. If there is a path from w to u, by definition $w \in C$ and $f[w] < f[u]$ as u was chosen to have maximal finishing time; if there is no path from w to u the key observation states that $f[w] < f[u]$. In either case this contradicts $f[w] > f[u]$. It follows that all nodes $v \in C$ are unvisited when we initiate the reverse edge depth-first search from u, and will thus all be visited in this search.

Having argued for the correctness of Alg. 6, the intuition behind it becomes a bit more clear. Claiming that a node v from which we can reach u can itself be reached from u (and thus is in the same strongly connected component) if and only if it has smaller finishing time allow us to make the right call for all nodes not reachable from u. The only problem is that there may be some nodes reachable from u with larger finishing times. However, one node in u's strongly connected component must have the largest finishing time, so for this node we happen not to make any wrong calls and u's strongly connected component is correctly identified from this node. As the algorithm essentially just consists of the two depth-first search traversals with a bit of simple book keeping added in, the time required by Alg. 6 is just $\mathrm{O}(|V| + |E|)$.

7 Maximum Flow

In several cases, the edge weights of a weighted graph can be considered capacities, with the obvious example being metabolic networks of biochemical reactions. Enzyme concentration will limit the throughput of most constituent reactions. To determine the maximum rate with which the network can produce a compound we will need to find the maximum flow of reactants we can push through the network, when each reaction is limited to its maximum throughput. There are also many non-biological situations where a problem can be well described as a network flow problem. However, probably the best reason for studying flow problems is the versatility of the problem, allowing other seemingly unrelated problems to be reformulated as maximum flow problems.

7.1 Flows, Cuts, and Augmenting Paths

The fundamental insight into solving maximum flow problems for a network is the equivalence between finding a maximum flow and finding a minimum cut of the network, as defined later in this section.

Definition 8 (Flow)**.** Let $G = (V, E, w)$ be a weighted network, and $s, t \in V$ distinct source and target nodes. A *flow* $f : V \times V \mapsto \mathbb{R}$ in G is a function satisfying

Edge constraints $\forall u, v \in V : f(u, v) \leq w(u, v)$, where we define $w(u, v) = 0$ for any $(u, v) \notin E$

Skew symmetry $\forall u, v : f(u, v) = -f(v, u)$

Flow conservation $\forall u \in V \setminus \{s, t\} : \sum_{v \in V} f(u, v) = 0$

The value of a flow is $|f| = \sum_{u \in V} f(s, u)$

This definition captures the intuition of a flow through a network of paths from s to t that never exceeds the capacity of any edge. The skew symmetry requirement may warrant a bit of explanation. If there is a positive flow $f(u, v)$ from u to v, we will mirror this by a negative flow of the same magnitude from v to u. This will be convenient in the following, as it captures where there are flows that can be cancelled. With the skew symmetry, the flow conservation requirement just states that there are no nodes, apart from s and t where the magnitude of the incoming flow and the outgoing flow is different, i.e. nodes where flow is either accumulating or being depleted. It is a straight forward exercise to show that the flow out of s, i.e. the value of the flow, equals the flow into t.

Definition 9 (Residual Network)**.** Let $G = (V, E, w)$ be a weighted network, and f a flow from s to t in G. The *residual network* of G and f is a weighted network $G_f = (V, E_f, c_f)$ where

- $(u, v) \in E_f$ iff $w(u, v) - f(u, v) > 0$
- $c_f(u, v) = w(u, v) - f(u, v)$ for $(u, v) \in E_f$

The residual network G_f basically captures the remaining, or residual, capacities once we are already sending flow f through the network. Notice that there may be an edge $(u, v) \in E_f$ even if $(u, v) \notin E$. However, this only happens if $(v, u) \in E$ and $f(v, u) > 0$. This corresponds to being able to *cancel* flow sent along an edge by f.

Definition 10 (Augmenting Path)**.** Let $G_f = (V, E_f, c_f)$ be a residual network for network $G = (V, E, w)$ and flow f. A path $\pi \equiv s = u_0 \rightarrow u_1 \rightarrow \ldots \rightarrow u_k = t$ containing no cycles is an *augmenting path* for G_f iff $\forall 1 \leq i \leq k : (u_{i-1}, u) \in E_f$.

The idea behind augmenting paths is that we can increase the flow along an augmenting path. Define flow f' such that $-f'(u_i, u_{i-1}) = f'(u_{i-1}, u_i) = \min_{1 \leq i \leq k}\{c_f(u_{i-1}, u_i)\}$, i.e. a flow along π with value equal to the minimum capacity of any edge on π. If we define the sum of flows in the obvious way, i.e. $(f + f')(u, v) = f(u, v) + f'(u, v)$, it is again an easy exercise to see that $f + f'$ is a flow in the original network G with value $|f| + |f'|$. The main concern is whether edge constraints are still satisfied, but this follows as $f'(u, v) \leq c_f(u, v) = w(u, v) - f(u, v) \Leftrightarrow f(u, v) + f'(u, v) \leq w(u, v)$ for all $u, v \in V$.

Definition 11 (s, t Cut)**.** Let $G = (V, E, w)$ be a weighted network, with distinct nodes $s, t \in V$. An s, t *cut* (or simply a cut) of G is a bipartition of V into sets S and $T = V \setminus S$ such that $s \in S$ and $t \in T$. The *capacity* of a cut is

$$c(S, T) = \sum_{u \in S} \sum_{v \in T} w(u, v) \tag{6}$$

The concept of an (s, t) cut is highly useful, as the capacity of any cut is an upper bound on the maximum flow in a network. If we extend the flow notation to include sets, such that

$$f(A, B) = \sum_{u \in A} \sum_{v \in B} f(u, v)\,, \tag{7}$$

then for any cut S, T we have

$$\begin{aligned} f(S, T) &= f(S, V) - f(S, S) && (8)\\ &= f(S, V) && (9)\\ &= f(\{s\}, V) + f(S \setminus \{s\}, V) && (10)\\ &= f(\{s\}, V) && (11)\\ &= |f|\,. && (12) \end{aligned}$$

Steps (8) and (10) follow from $f(A, B) + f(A, C) = f(A, B \cup C)$ if B and C are disjoint (just split the sum over nodes in the union), step (9) follows from $f(A, A) = 0$ for any A (by skew symmetry, as $f(u, v)$ is a term in the sum iff $f(v, u)$ is a term in the sum), and step (11) follows from flow conservation (as $f(\{u\}, V) = 0$ for $u \in V \setminus \{s, t\}$, hence $f(A, V) = \sum_{u \in A} f(\{u\}, V) = 0$ for $A \subseteq V \setminus \{s, t\}$). We also have

$$\begin{aligned} f(S, T) &= \sum_{u \in S} \sum_{v \in T} f(u, v) && (13)\\ &\leq \sum_{u \in S, v \in T m f(u,v) \geq 0} f(u, v) && (14)\\ &\leq c(S, T) && (15) \end{aligned}$$

where step (14) follows from the edge constraint on f. It follows that $|f| \leq c(S,T)$ for any flow f and cut S,T.

Theorem 1 (Maximum flow, minimum cut)**.** *Let f be a flow in network $G = (V, E, w)$. Then the following conditions are equivalent:*

1. *f is a maximum flow in G*
2. *G_f has no augmenting path*
3. *$|f| = c(S,T)$ for some cut S,T*

Proof. If G_f has an augmenting path, we can define a positive flow f' along this path such that $f + f'$ is a flow in G with $|f + f'| = |f| + |f'| > |f|$. So clearly f being a maximum flow implies that G_f has no augmenting path. We have also just seen that no flow can have value larger than $c(S,T)$ for any cut so $|f| = c(S,T)$ for some cut must imply that f is a flow with maximum value. It remains to argue that the non-existence of an augmenting path implies a cut S,T with $|f| = c(S,T)$.

Let $S = \{u \in V \mid \text{there is a path from } s \text{ to } u \text{ in } G_f\}$, i.e. S is the set of nodes we can reach from s in G_f. Clearly $t \notin S$, as otherwise we would have an (augmenting) path from s to t in G_f, so $S, T = V \setminus S$ is an s,t cut. Moreover, we have $\forall u \in S, v \in T$: $0 = c_f(u,v)$, as otherwise there would be an edge in G_f from a node $u \in S$ to a node $v \in T$, contradicting that v is not reachable from s. Hence, $0 = \sum_{u \in S, v \in T} c_f(u,v) = \sum_{u \in S, v \in T} w(u,v) - f(u,v) = c(S,T) - f(S,T)$ and $|f| = c(S,T)$ follows. ☐

Algorithm 7 Generic Augmenting Path Maximum Flow

Input: Network $G = (V, E, w)$ and distinct source and sink nodes $s, t \in V$
  Initialise residual network $G_f = G$ and flow f to $\forall u, v \in V : f[u,v] = 0$
  while there is simple augmenting path π in G_f **do**
    Define flow f' along π such that minimum capacity of edges is fully utilised
    Set $f = f + f'$ and update G_f accordingly

By repeatedly updating the flow along an augmenting path in the residual network, one would imagine that we can thus find a maximum flow. This is captured in Alg. 7. However, the efficiency – and even termination – of this algorithm may depend crucially on how the search for an augmenting path is performed. If we use breadth-first search, the algorithm will perform at most $\mathrm{O}(|V||E|)$ iterations. The most time consuming step in each iteration will be the search for an augmenting path, which requires time $\mathrm{O}(|V| + |E|)$. This is known as the Edmonds-Karp algorithm and requires time at most $\mathrm{O}\left(|V||E|^2\right)$. Rather than going into a formal proof of this complexity, we will proceed to discuss a method based on *preflows* requiring only $\mathrm{O}\left(|V|^2|EE|\right)$ time.

7.2 Preflows

The most efficient algorithms for solving the maximum flow problem and related problems is based on the preflow-push approach. In this section we will cover a relatively straight forward version of this approach that still allows maximum flows to be computed fairly efficiently. Assume that we just start pumping whatever stuff it is our network transports out from s at the maximum capacity of each outgoing edge. Assume further that our system is capable of adjusting, so that if there is a way to shift flows to routes with spare capacity leading to t this will happen. When the system reaches equilibrium, there may be nodes before the minimum cut that receives more flow than they can get rid off, and are thus overflowing; beyond the minimum cut, nodes will be able to pass the flow they receive onward to eventually reach t. So all we need to do is to pass word of the overflows back towards s to have the flow produced from s reduced accordingly. This is the intuition behind the preflow-push methodology.

Definition 12 (Preflow)**.** Let $G = (V, E, w)$ be a network and s and t distinct source and target nodes. A *preflow* $f : V \times V \mapsto \mathbb{R}$ in G is a function that satisfies edge constraints and skew symmetry as in Def. 8, but with flow conservation relaxed to

No flow depletion $\forall u \in V \setminus \{s\} : \sum_{v \in V} f(u, v) \leq 0$

The excess flow into node $u \in V$ is $e_f(u) = \sum_{v \in V} f(v, u)$. A node $u \in V \setminus \{s, t\}$ is said to be *overflowing* if $e_f(u) > 0$.

The idea is to start with a preflow just sending flow to s's neighbours utilising the full capacity of the outgoing edges from s. This is then attempted shifted all the way to t, and once this fails excess flow is returned to s. To systematically do this, we will further need to keep an adjustable height for each node.

Definition 13 (Height Function)**.** Let $G = (V, E, w)$ be a network and f a preflow from s to t in G. A *height function* $h : V \mapsto \mathbb{N}$ consistent with f is a function satisfying

- $h(s) = |V|$
- $h(t) = 0$
- If $f(u, v) < w(u, v)$ then $h(u) \leq h(v) + 1$

If we consider the residual network G_f of preflow f, a height function thus guarantees that there are no edges with a drop of more than 1. It also follows that there cannot be a augmenting path from s to t in G_f if f is a preflow with a consistent height function. This follows as the height between consecutive nodes on a path in G_f drops by at most 1, so to go from s at height $|V|$ to t at height 0 we would need to be nodes with heights $\{0, 1, \ldots, |V|\}$ on the path. But that would require $|V| + 1$ distinct nodes.

If we initialise the preflow by saturating edges out of s, as already described, and set the height of every node except s to 0 and the height of s to |V|, these will be compatible: only s sits at a height more than 1 above any other node, but as the preflow fully utilises the outgoing edges of s it will have no outgoing edges in the residual network. If we can maintain a consistent preflow and height until the preflow becomes a proper flow, we must have a maximum flow by Theorem 1 as the residual network of this flow will have no augmenting path.

Algorithm 8 Push Method

```
define Push(u, v, a):
    f[u, v] = f[u, v] + a, f[v, u] = −f[u, v]
    e[u] = e[u] − a, e[v] = e[v] + a
```

Algorithm 9 Lift Method

```
define Lift(u):
    h[u] = min{h[v] + 1 | (u, v) ∈ E_f}
```

Algorithm 10 Generic Preflow-Push

Input: Network $G = (V, E, w)$ and distinct source and sink nodes $s, t \in V$

```
 1: Initialise preflow f to f[s, u] = w(s, u) for (s, u) ∈ E and f(u, v) = 0 otherwise
 2: Initialise overflow e to e[u] = w(s, u) if (s, u) ∈ E and e[u] = 0 otherwise
 3: Initialise height to h[s] = |V| and h[u] = 0 otherwise
 4: Initialise G_f to residual network of G and f
 5: while there is overflowing node u do
 6:     if ∃v : h[v] = h[u] − 1 and c_f(u, v) > 0 then
 7:         if c_f(u, v) ≤ e[u] then
 8:             Push(u, v, c_f(u, v))
 9:         else
10:             Push(u, v, e[u])
11:         Update G_f according to new f
12:     else
13:         Lift(u)
```

This is implemented by Alg. 7. If a node is overflowing we push excess flow to a lower node; if there is no lower node we can push flow to, we lift the node up to a level such that there is one. It terminates when no nodes are overflowing, i.e. when f has become a proper flow. We still need to argue that it maintains a consistent preflow and height, and that it will terminate. To see that the preflow and height function remain consistent,

observe that a Lift operation will never increase the height of a node to more than 1 above any node it has an edge to in E_f. So Lift operations do not break consistency. A Push operation does not change any heights, but may create a new edge (v, u) in the residual network. However, this only happens if $h[v] = h[u] - 1$ so the change in height along this new edge will be 2 more than the minimum required.

If a node u is overflowing, there must be a path in G_f from u to s – the overflow must eventually be traceable back to s, resulting in edges in the reverse direction of the flow. It follows that we can never lift a node above height $2|V| - 2$ as there would otherwise be an edge on the path from u to s in G_f where the height drops by more than 1 after the lift. When we lift a node, its height increases by at least 1 – an overflowing node will have at least one outgoing edge, but as it has no neighbours at a lower level than itself $\min\{h[v] + 1 \mid (u, v) \in E_f\} \geq h[u] + 1$. So Alg. 10 can never perform more than $\mathrm{O}\left(|V|^2\right)$ Lift operations in line 13.

For the (saturating) Push operations in line 8, observe that this operation removes the edge (u, v) from G_f. This edge can only reappear as a consequence of a Push operation along (v, u). Due to the restriction that pushes are only performed downhill, the height of u must have increased by at least 2 before the next time we push flow along (u, v). By the limit on the maximum height, for each edge we can apply a saturating Push operation at most V times in each direction. It follows that the total number of saturating Push operations is at most $\mathrm{O}\left(|V||E|\right)$ times in total.

This leaves the (nonsaturating) Push operations of line 10. To bound the number of these, consider the potential function $\Phi = \sum_{u \in V, e[u] > 0} h[u]$, i.e. the sum of heights of overflowing edges. Clearly, this is always non-negative. A nonsaturating push will always decrease Φ by at least 1. The overflowing node u will not be overflowing after the push, and the target node of the push that may go from not being overflowing to being overflowing is at a height level 1 lower. The total increase in Φ from Lift operation in line 13 is $\mathrm{O}\left(|V|^2\right)$. A Push operation in line 8 can at most increase Φ by $2|V| - 3$, as the target node may change from non-overflowing to overflowing with the source not still overflowing. It follows that Lift operations and saturating Push operations contribute a total increase of Φ of $\mathrm{O}\left(|V|^2 + |V||V||E|\right)$, resulting in a bound on nonsaturating pushes of at most $\mathrm{O}\left(|V|^2|E|\right)$ (assuming $|V| = \mathrm{O}\left(|E|\right)$). This is also the overall time complexity, which is not the most efficient version of the Preflow-Push method but should be sufficient in most cases.